1995

Synchronization Expressions In Parallel Programming Languages

Lifu Guo

Synchronization Expressions in Parallel Programming Languages

by

Lifu Guo

Department of Computer Science

Submitted in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy

Faculty of Graduate Studies
The University of Western Ontario
London, Ontario
December 1994

ISBN 0-315-99255-7

Canada

ABSTRACT

In this thesis, we first review current trends in the areas related to parallel programming languages. By reviewing data-parallel and control-parallel paradigms, parallelizing compilers and parallel programming languages, we demonstrate the necessity of developing high-level parallel programming languages.

Synchronization is an essential part of parallel processing. However, many synchronization mechanisms currently used in parallel programming languages are at too low a level to be compatible with the other constructs of high-level languages. We propose and develop new constructs for synchronization termed "synchronization expressions". These new constructs relieve programmers of the burden of imposing synchronization, requiring them only to specify the necessary constraints. The new constructs demand no structural changes to the base language and allow complicated synchronization constraints to be expressed easily.

We also introduce a new family of languages, called synchronization languages, to model the synchronization of parallel processes. A synchronization language is a restricted regular language. Synchronization languages provide us with not only a theoretical model for the semantics of the synchronization of multiprocessors but also a systematic method for implementing synchronization expressions. We also discuss the idea of using reversed alternating finite automata in the implementation of synchronization expressions.

In Memory of My Father

ACKNOWLEDGEMENTS

I am indebted to many people for their help and support during the years I have worked on my Ph.D program. First of all, I wish to thank my supervisor, Prof. S. Yu, for the superior help provided in suggestions and discussions concerning my thesis work and the research. Without him, this work would have been impossible.

I would like to thank my wife, Ping, for her incredible patience and understanding.

I also would like to thank the members of my advisory committee: Prof. H. Jürgensen, for his invaluable suggestions and help, and Prof. J. M. Bennett, for his invaluable criticisms.

Special thanks to Dr. K. Salomaa for his cooperation in our research. He was of great assistance in the development of synchronization languages. Several reviewers read my thesis in detail and offered invaluable advice. I also would like to thank Mr. P. Scheyen for his assistance in debugging the ParC compiler. His help was crucial in making the compiler work. I thank these people as well for their help: Dr. R. Govindrajan, McGill University, Prof. P. Wang, Kent State University, and Prof. D. Padua, University of Illinois.

Thanks are due to Prof. J. Barron, Prof. P Streufert, and Mr. A. Benn for their careful proof-reading of this thesis.

Finally, I would like to thank my employers, The University of Western Ontario and IBM Toronto Laboratory, for their support.

TABLE OF CONTENTS

Chapter 1

Introduction

It has become clear that traditional mainframe computing systems lack the ability to address the various needs of today's computation adequately. For example, in database systems, steady increases in the size of databases and more complex query applications have brought traditional computing systems to their performance limits. Since the early eighties, an increasing number of parallel systems have been designed and built, and many of them have been made commercially available. Thus, there is a growing demand for user-friendly and effective parallel programming languages that can utilize the resources of parallel systems. Many attempts have been made to meet this demand. However, the development of parallel programming languages still lags behind the development of parallel architectures.

Synchronization is an essential part of parallel computation. Many synchronization mechanisms have been proposed, including semaphores [37], conditional critical sections [65], monitors [66], path expressions [25, 26, 27], and Linda constructs [6, 28]. For example, the Fortran in the Cedar project [97] supports semaphore-based mechanisms for synchronization. Monitors have been used in Concurrent Pascal [57] and Modula [121], and path expressions have been implemented in Path Pascal [26]. Attempts have been continually made to develop new language constructs for expressing synchronization. For instance, in the parallel programming language Jade [80], new language constructs *withth*, *with*, *withonly*, and shared data objects associated with a synchronization type called token, can be used to express synchronization constraints.

It has been recognized that, however, synchronization mechanisms used in most parallel programming languages are either at too low a level to be compatible with other constructs of high-level languages or too complicated to be used for expressing synchronization constraints easily. As more and more intricate parallel algorithms need to be implemented, the complexity of synchronization will soon exceed a programmer's ability to handle synchronization with low-level constructs. Clearly, there is a demand for simple, user-friendly, and efficient high-level constructs for synchronization. In this thesis, statement tags and synchronization expressions are proposed to meet this demand. A statement tag, as may intuitively evident, is a label in front of a statement that identifies the execution of the entire statement. A synchronization expression (SE) is an expression of statement tags, and it is used to specify synchronizations between processes. Statement tags and synchronization expressions have been implemented in a new parallel programming language, *ParC* (Parallel C).

ParC has the following features:

1. Shared variables have scopes; they can be declared at any level of a program and their scope rule follows that of the variables in C.

2. Statements **pexec**, **pfor**, and **sfor** are provided for the explicit expression of parallelism. These statements are intended to free the programmers from the burden of process management.

3. Statement tags are introduced to identify the execution of a statement. They are used for specifying synchronization. Statement tags can be of scalar or array types.

4. Synchronization expressions (SEs), expressions of statement tags, are used to specify synchronization between processes. The purpose of introducing SEs is to have synchronization specified directly, so that users need only specify *what* has to be synchronized rather than *how* to perform its synchronization.

The necessity of a formal semantic definition for SEs is apparent both in theory and in practice. For example, it is essential to be able to test whether two SEs are equivalent. The compiler should be able to check whether two SEs contradict each other and to simplify SEs. All of these tasks rely on a formal semantic definition of SEs that models our intuitive and informal description of SEs. As well, this definition can be expected to provide a systematic way of implementing the synchronization

controls specified by SEs. For these purposes, we introduce a new family of languages, called synchronization languages, to develop a semantic model for synchronization expressions [50]. A synchronization language is a restricted regular language. In formal language theory, the family of synchronization languages is a proper subset of the family of regular start-termination languages (st-languages), where an st-language is (informally) a subset of the shuffle of the language $(a_{1s}a_{1t})^*, ..., (a_{ns}a_{nt})^*$. Note that there are finite st-languages that are not synchronization languages. In fact, synchronization languages describe exactly how SEs are being implemented in the parallel programming language *ParC*.

In the remainder of this chapter, a brief introduction to parallel programming paradigms will provide the reader with the necessary background in parallel programming. The advantages and disadvantages of parallelizing compilers and parallel programming languages will be considered, and issues related to data parallelism and control parallelism will be discussed.

Chapter 2 gives an overview of the existing synchronization primitives in programming languages, including semaphores, critical regions, monitors, and path expressions. We also look at how the high-level synchronization primitives in Ada and the intermediate-level synchronization primitives in Linda are used to solve synchronization problems.

Chapter 3 sets out the motivation and objective in the development of *ParC* and provide an overview of the *ParC* language. Then we focus on synchronization expressions (SEs). The syntax and a description of SEs are given in this chapter. Examples are provided to show how SEs can be used to solve synchronization problems in parallel programming. As well, we give a detailed discussion of the advantages of SEs over other synchronization primitives.

Chapter 4 is devoted to synchronization languages and the theoretical model used to describe the semantics of SEs. We review current theoretical models for parallel processing, and demonstrate why it is necessary to study synchronization languages. The semantics of SEs are formally described in terms of synchronization languages in this chapter. Relevant properties of synchronization languages are also studied.

Chapter 5 describes how reversed alternating finite automata can be used in the implementation of synchronization expressions.

Conclusions and directions for future research are given in the final chapter.

1.1 Parallel Programming Overview

Conventional sequential processing has a commonly accepted model for designing and analyzing algorithms. The RAM model consists of one central processing unit with a random-access memory attached to it. It has been successfully used to evaluate the performance of sequential algorithms. In contrast, parallel processing suffers from the lack of such a widely accepted model, not only for designing and analyzing algorithms but also for constructing programming languages. Although many researchers have used the PRAM model in the design and analysis of parallel algorithms, the PRAM model is not generally accepted in parallel processing because it does not take into account synchronization and communication costs. There is no widely accepted model, primarily because the performance of parallel computations depends on a set of complex interrelated factors that are machine-dependent. These factors include process allocation, scheduling, communication, and synchronization.

Current parallel programming models closely depend upon the underlying parallel system architecture. The shared-memory model uses shared memory for inter-process communication and synchronization. The message-passing model uses messages passed among processes for inter-process communication. Recent developments in both software and hardware technology, such as the KSR multiprocessor, tried to take advantage of both shared-memory multiprocessors and distributed-memory multiprocessors. Such multiprocessors with physically distributed and logically shared memory allow users to program on a distributed-memory system using single-address memory space. As a result, the shared-memory programming model has attracted much attention recently.

In research related to parallel programming languages, much effort has been devoted to developing parallelizing compilers that can automatically detect parallelism in sequential programs. Much work has also been devoted to developing new programming languages that are convenient for writing explicitly-parallel programs.

In the following section, we review some basic concepts of parallel system architectures in order to provide the necessary background for a further discussion of parallel programming languages. We also comment on what can and cannot be achieved by using parallelizing compilers and explicit parallel programming languages. Finally, we discuss the two paradigms of parallel programming: data-parallel and control-parallel approaches.

1.1.1 Parallel Computer Architectures

We have witnessed the emergence of a wide variety of new computer architectures for parallel processing in the past decades. Flynn's taxonomy [41] classifies architectures according to the presence of single or multiple streams of instructions and data. This yields the four categories below:

- *SISD* (single instruction stream and single data stream) defines serial computers. This class comprises all conventional uniprocessor systems.
- *MISD* (multiple instruction stream and single data stream) involves multiple processors applying different instructions on a single data stream; they are also known as systolic arrays, providing pipelined execution of parallel algorithms.
- *SIMD* (single instruction stream and multiple data stream) involves multiple processors simultaneously executing the same instruction on different data.
- *MIMD* (multiple instruction stream and multiple data streams) involves multiple processors autonomously executing diverse instructions on diverse data.

Although Flynn's taxonomy is insufficient to classify various modern computers [39], it provides a useful shorthand for characterizing architectures. The *SIMD* and *MIMD* classifications of parallel computers are still the most generally used taxonomy of parallel computer architectures.

Current parallel programming has been strongly attached to target machine architecture. Such as shared-memory and message-passing parallel programming models. *SIMD* and *MIMD* modes of programming have been studied separately. Recently, the *SPMD* (single program stream and multiple data streams) mode of parallelism has attracted considerable attention. The *SPMD* mode of parallelism is a type of *MIMD* mode in which the processors are all restricted to execute the same program (asynchronously with respect to each other).

SIMD Architecture

In an *SIMD* computer, many processors simultaneously execute the same instructions, but on different data. Examples of *SIMD* machines with massive parallelism are Connection Machine (CM-2) and MasPar (MP-2). Typically, *SIMD* machines are built with simple *processing elements* (PEs), but compensate for this simplicity

by using many of them together. The processing component of an *SIMD* computer consists of one control unit (CU) and a large number of PEs. For example, MP-2 has up to 16,384 PEs. In an *SIMD* machine, a single master processor broadcasts program instructions to the individual PEs, which carry out the instructions on the data that is stored locally on each PE. PEs can be disabled temporarily so that operations are only carried out by specific PEs; this provides a way of making computations data-dependent, to simulate conditional statements that exist in all languages. PEs can also transfer data among themselves. In MP-2, interprocessor communications are handled by two separate mechanisms, the X-net and the global router, depending upon which is more appropriate for a given application. For situations in which an entire array of data is to be moved across the PE lattice, the X-net communication mesh is used. Random communications between arbitrary processors are possible via a three-stage global router, which emulates a crossbar switch.

The lock-step nature of *SIMD* machines provides one significant advantage for programming: the user does not need to worry about synchronization. Experience has shown that *SIMD* machines are successful in solving a wide range of scientific problems in such diverse areas as image processing and DNA sequence analysis. Because an important, but limited, class of problems fits this model extremely well, there is some impetus for the design and construction of these machines. Clearly, however some large sets of problems do not lend themselves to efficient execution in a *SIMD* architecture. The operations required for such problems cannot be easily organized into repetitive operations on uniformly structured data. Instead, these operations tend to be unstructured and unpredictable. Their addressing patterns tend to be data-dependent, so the architecture cannot easily preload data by anticipating future access. *MIMD* architecture, considered to be a more general-purpose parallel architecture in parallel processing, can handle such demands more easily.

MIMD Architecture

In terms of the relationship of processors to memory, MIMD systems can be divided into two categories: those that contain physical memory shared by all processors, called shared-memory systems, and those that do not, called distributed-memory systems. To take advantage of both shared-memory systems and distributed-memory systems, software is used to provide virtual shared memory space to programmers in

a distributed-memory architecture.

Shared-Memory Multiprocessors

A multiprocessor with a physical shared memory has a single, global, shared address space visible to all processors. Any processor can read or write any word in this address space, and communication between processors takes place via the shared memory. There are two basic ways to build a shared-memory multiprocessor. The first way is to put all the processors on a single bus along with memory modules. The second way is to use switching networks to connect processors and memory modules.

In a bus-based shared-memory multiprocessor, a processor makes read/write requests over the bus. Each processor, in addition, can have a cache memory. The cache memory enables the processors to reduce their use of the shared bus, thereby limiting the effects of contention on the bus performance when processors have to go to shared memory through the bus. However, the cache memory cannot fully solve the bus contention problem. The bus-based shared-memory multiprocessors offer the simplest topology but have the highest potential for contention. This prevents bus-based systems from having a large number of processors.

The Sequent Symmetry computer, a bus-based, cache-coherent machine, has a single global memory attached to the bus. Each processor has a local cache, the coherence of which is maintained by hardware. In the system released in 1987, the system bus has a bandwidth of up to 53.3 MB/sec. The system uses Intel 80386 processors running at 20 MHZ. The 64KB local cache attached to each processor is used to reduce the need for bus bandwidth. The recently released Symmetry 2000 series uses the same bus to support up to 30 Intel 80486 processors running at 25 MHZ, each with a 512 KB local cache. The Silicon Graphics Iris multiprocessor workstation is another bus-based, cache-coherent system. The Iris supports up to eight 40 MHZ MIPS R3000 processors on a bus with 64 MB/sec bandwidth. Each of the processors has a 64 KB first-level cache and a 1 MB second-level cache.

There are many kinds of interconnection network developed for shared-memory multiprocessor systems, including shuffle-exchange networks, combining networks, and crossbar interconnections networks. To access a shared memory, a processor sends a request to the appropriate memory module via the switching network, and the memory module sends the reply back via the network.

In large-scale shared-memory systems like the BBN Butterfly family of multiprocessors, each processor has a local memory, and all the local memories form the shared memory. The original Butterfly machine (Butterfly I) consists of up to 256 nodes connected by a 4 MB/sec (per link) Butterfly switch. Each processor node contains an 8 MHZ MC68000.

Shared memory offers many advantages, including an uniform address space and referential transparency. The uniform address space frees programmers and compilers from the burden of data distribution and permits fine-grained data sharing. Referential transparency ensures that object names (i.e. addresses) and access primitives are identical for both local memories and remote memories. Furthermore, a uniform address space simplifies programming, compilation, and load balancing, because the same code runs on all processors. Most of the techniques developed for multi-tasking time-sharing computers can be used directly on shared-memory multiprocessors. Shared-memory multiprocessors have been successfully employed to exploit a moderate amount of parallelism. However, since such architectures are not scalable, it is not possible to build a shared memory system that allows exploitation of a massive amount of parallelism. As the number of processors trying to access the shared memory increases, memory access contention increases. Eventually, the access to shared memory becomes a bottleneck limiting the speed of the computer. The use of local cache memory can alleviate this problem by permitting commonly used data to be stored locally on each processor. However, the bottleneck still exists because the cache memory cannot eliminate all the requests for access to the shared memory.

Distributed-Memory Multiprocessors

In a distributed-memory multiprocessor, each processor has its own private memory, without any global shared memory space. Communication between processors takes place by passing messages. The problem of how to link these processors together still remains. Connecting them all to a single bus, or through a single switch, would lead to the same sorts of bottleneck as in shared-memory systems.

Among the various interconnection schemes for distributed memory systems, hypercube interconnection is a popular topology. With hypercube interconnection, each of the $N = 2^n$ processors is connected to n neighbors. This allows hypercube interconnection with high scalability. Figure 1.1 shows some multidimensional hypercubes.

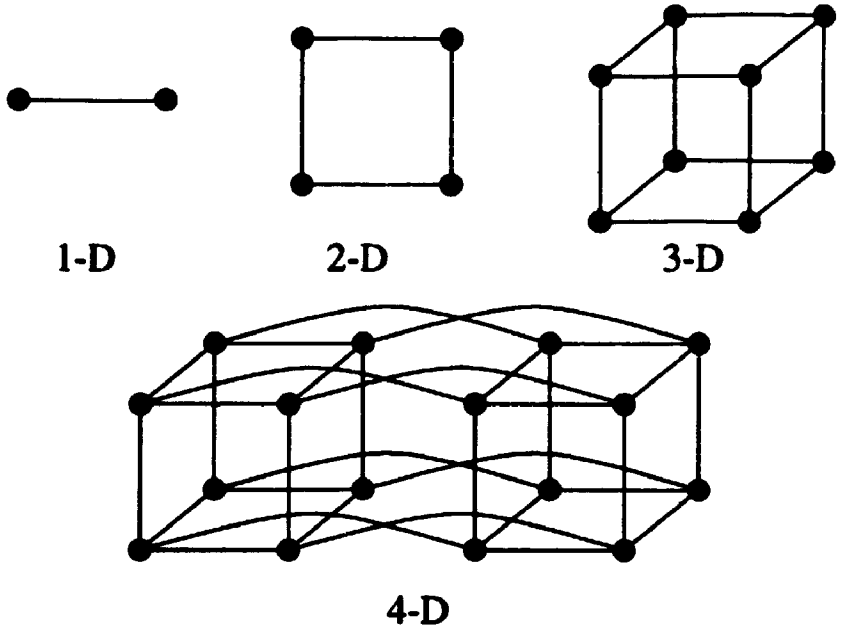


Figure 1.1: 1-D to 4-D hypercube

Hypercubes have several advantages. First, the number of nodes in a nypercube grows exponentially with the number of connections per node, so that a small change in the hardware at each node allows a large increase in the size of the computer. Second, the number of alternative paths between nodes increases with the size of the hypercube. This helps relieve congestion. Third, efficient algorithms are known for routing messages between processors in a hypercube (e.g. random two-phase routing). As a result, the hypercube architecture has attracted significant research interest.

In general, programming on distributed-memory systems is more difficult than on shared-memory systems. The message-passing models force programmers to be conscious of data movement between processors at all times. Also, since data in the message-passing model passes between multiple address spaces, it is burdensome to pass complex data structures.

Commercially available *MIMD* systems with hypercube architecture are the Intel iPSC and nCUBE systems. The Intel iPSC/860 is scalable from 8 to 128 nodes, and the nCUBE 2S can have up to 8192 nodes.

Virtual Shared-Memory Multiprocessors

Many architectures have been proposed to capture the desirable properties of both shared-memory systems and distributed-memory systems. Most of this research

attempts to simulate a shared memory on a distributed-memory system, i.e. a system with distributed memory shares a single address space. This is called a virtual-shared memory system.

A logically shared address space, whether supported by hardware or software, greatly simplifies programming. As the number of processors increases in parallel computers, physically distributing the memory modules allows each processor faster access to memory regions that reside closely to the processors. The combination of a logically shared address space and a physically distributed memory is a new trend in the design of massively parallel and high-performance computers, including such machines as the Kendall-Square Research Corporation's KSR-1 multiprocessor. Even machines such as Intel Corporation's Paragon multiprocessor, which requires explicit message-passing in the tradition of distributed-memory machines, have software support to provide a shared address space.

The simplest form of distributed shared memory is to divide all shared memory into fixed-size pages, with each page residing on exactly one processor. Processor references to a local page are made by hardware in the usual way. However, referencing a remote page causes a fault and a trap to the operating system. In a virtual shared memory system, the operating system fetches the page just as it would in a traditional virtual memory system. In a virtual memory system, the page is fetched from the disk; but in a virtual shared-memory system, however, the page is fetched from another processor's memory. Many algorithms have been developed to improve the performance of such distributed shared-memory systems.

The primary advantage of distributed shared-memory over message passing is the simple abstraction of the memory system that is provided to the programmer, an abstraction that can be easily understood. The access protocol used is consistent with the way sequential applications access data, allowing a more natural transition from sequential to distributed applications. In principle, a parallel computation written for a shared-memory system can be executed on a distributed shared-memory system without change. Similar to shared-memory systems, distributed shared-memory hides the remote communication mechanism from the processes and allows complex structures to be passed by reference, substantially simplifying the programming effort. Moreover, the distributed shared-memory architecture takes advantage of a distributed memory system's high scalability. However, the virtual shared memory introduces high overheads.

1.1.2 Data-Parallel and Control-Parallel

The tradeoff between the cost and benefit of parallel programming needs to be considered. Obviously, benefits include the fast execution of applications due to the parallelism. The costs include the overhead of process management, synchronization and communications. Most people consider parallel programming to be more difficult than sequential programming.

For example, the overhead of process management follows from the decomposition of an application so that it may be run in parallel. The choice of coarse-grain or fine-grain parallelism can be a concern. Synchronizations also often introduce significant overhead; the type of available synchronization primitives in particular can be a significant factor in deciding the complexity of parallel programming. These issues can be addressed differently according to various parallel computer architectures. For example, distributed-memory systems can provide facilities for data distribution; *MIMD* machines need to provide mechanisms for synchronization properly; the specification of fine-grain parallelism is one of the major concerns in *SIMD* machines.

There are two general paradigms in parallel programming: data-parallelism and control-parallelism. They are the two basic methods that are employed in programming on parallel systems:

- Data-parallel: this method assumes that there is a large data set that needs to be processed and that there is a single process for each data element in the set. The same set of instructions is applied to each element in the data set.

- Control-parallel: this method assumes that there are separate, relatively independent processes that can be executed simultaneously. Each separate process is then assigned to a separate CPU, which is entirely dedicated to that particular function or process.

Data-parallelism, which is often associated with the *SIMD* architecture, is the natural paradigm for a large percentage of problems in science and engineering. Its synchronous pattern of execution means that there is only one control: race conditions and deadlocks cannot occur. Data-parallelism is achieved through the simultaneous execution of the same operation across a set of data. In contrast, control-parallelism is achieved through the simultaneous execution of different operations. In order to take advantage of parallel hardware, one must exploit potential for the data- and/or control-parallelism inherent within an application. Advocates of the data-parallel

approach argue that the amount of inherent control-parallelism in an application is a fixed number independent of the size of the data set, whereas data-parallelism, by definition, increases with the size of the problem to be solved.

To illustrate the difference between the data-parallel and control-parallel approaches to parallel computation, let us consider how to parallelize the classical prime-finding algorithm, the Sieve of Eratosthenes. We chose the following example from [60]. A prime number has exactly two factors: itself and 1. A number is composite if it is not prime. The Sieve of Eratosthenese begins with the list of natural numbers 2, 3, 4, ..., N, and then gradually weeds composite numbers from the list by marking multiples of 2, 3, 5, and successive primes. The time complexity of the algorithm is $\Omega(N)$, making it impractical for determining the primality of "interesting" numbers—those with hundreds of digits—because N increases exponentially with the number of digits. However, exploiting parallelism in the algorithm can lead to practical solutions of the prime number problem.

Control-Parallel Solution

What is a control-parallel solution to the prime number problem? Suppose there are a number of processes working together to do the multiplication of primes. One process could be marking multiples of 2, while another process is marking multiples of 3, and yet another process is marking multiples of 5, and so on. Even though each process is executing the same set of instructions, this approach is control-parallel, because each process executes instructions at its own rate.

Often the processes in control-parallel algorithms have complicated interactions. What are the interactions in this control-parallel solution? The parallel sieve algorithm has two trouble spots: two or more processes may try to grab the next available prime number, and a process may take as prime a number that is actually composite, but has not been struck yet. Both of these problems are caused by race conditions. Fortunately, neither race condition will lead to an incorrect solution; they merely lead to duplicated work. The algorithm designer can address these problems by introducing critical sections, or simply ignore the problems, and accept the possibility of wasted work. In either case, however, performance suffers. Another problem with the control-parallel approach to the Sieve of Eratosthenes is that the amount of work performed by various processes varies with the size of the prime number. For example, the process marking multiples of 2 must mark every second list element, while

the process marking the multiples of 23 performs less than 10% as much work. Such a severe load imbalance makes it difficult for the algorithm to achieve s high speedup, even with large numbers of processes.

Data-Parallel Solution

Let us consider a pure data-parallel solution to this problem. The fundamental unit of parallelism in this algorithm is the natural number. Therefore, we assume there is a process associated with each number. These processes work together in lock step to strike out prime multiples. In the first iteration of the loop, all processes associated with multiples of 2 between 4 and N set the primality flag of their number to FALSE. In the second iteration, processes representing all multiples of 3 ranging from 9 to N set the primality flag to FALSE. At any point in the execution of the algorithm the processes eliminate the multiples of exactly one prime.

The primary weakness of the pure data-parallel approach is that it assumes an unlimited resource of processors. Although the assumption makes the algorithm easier to design, it leads to poor performance. In reality, the number of processes is always bounded. We can map a group of natural numbers with each process, and each process can perform the operations on that group of numbers.

The sequential and control-parallel versions of the sieve algorithm take advantage of the fact that the multiples of prime value v can be eliminated by striking out every vth element of the array, beginning with value v^2. In contrast, the data-parallel algorithm requires that the computer check every natural number every time a new prime is found. As a result, the pure data-parallel algorithm performs far more calculations than the sequential algorithm.

The control-parallel approach is characterized by a relatively small number of asynchronous processes. Even if all processes are executing the same program, they may be executing different instructions at any given moment. The data-parallel approach is characterized by a relatively large number of synchronous processes executing a single instruction stream. One must properly define the fundamental unit of parallelism in order to execute a data-parallel algorithm efficiently on real hardware.

What makes an algorithm parallel is lots of threads of control, not lots of data. Data-parallelism is a convenient paradigm for expressing control parallelism in certain domains. The parallelism is realized, however, by inferring control-parallelism from

data-parallel code and executing a control-parallel program. For some applications, it is essential to exploit both styles of parallelism within the same program. In particular, for signal processing applications, physical considerations often make it impossible to increase the size of the data sets arbitrarily. Thus, it may be impossible for a single data-parallel thread of control to use all nodes effectively. Depending on the size of the machine and the data sets, the same source program can benefit from different styles of parallelism: data-parallelism, control-parallelism, or some combination of the two.

A simple example illustrates how the data-parallel approach cannot be used to express all parallelism. From this simple example, more complicated and practical examples can be derived. Let s_1 and s_2 be two statements in a program, and

s_1: `a = b + c;`

s_2: `d = e + f + g;`

Obviously, the statements s_1 and s_2 are independent, and can be executed in parallel. Such parallelism can be expressed easily using the control-parallel approach. However, trying to express this in a data-parallel fashion can only lead to a sequential execution of the statements. In the data-parallel approach, one process first executes the statement s_1, while other processes are suspended. The next step involves a second process to execute the statement s_2, while all the other processes are idle. The data-parallel solution, then, can lead to poor performance, since parallelism cannot be utilized.

1.1.3 Parallelizing Compilers and Parallel Programming Languages

Programming parallel computers is widely recognized as being more difficult than programming sequential computers, but much of the blame can be traced to the programming languages used. Since current *SIMD* machines use the data-parallel programming paradigm, conventional applications need to be rewritten to be executed efficiently on target machines. In addition, only limited types of applications can take advantage of *SIMD* architectures. In contrast, *MIMD* machines are considered to be more general purpose. These *MIMD* architectures have been attracting more attention regarding software research and development than SIMD machines. In this section, we examine a variety of ways to program *MIMD* parallel machines,

using imperative programming languages. First, we consider programming with a conventional sequential language, followed by the use of a parallelizing compiler to automatically detect parallelism in the sequential code. Then we consider the approach commonly used in present commercial parallel systems, i.e., conventional programming languages extended with low-level machine-dependent system calls. Finally, we look at the advantages of new programming languages that are being developed for parallel processing and conventional programming languages that are extended with high-level parallel constructs. These languages or language constructs are high-level, machine-independent constructs. Programmers can express parallelism, synchronization, and communication, etc., at a level of abstraction compatible with other high-level-language constructs.

Parallelizing Compilers

A major impediment to the wide acceptance of parallel machines is the complexity of writing efficient parallel programs. Although most parallel machines support variations of Fortran and C, the many features of different parallel machines result in distinct, architecture-specific extensions to the languages. In order to achieve efficient program execution on a parallel machine, the programmer must first become acquainted with the programming paradigm dictated by the architecture of the target machine. Programming with architecture-specific extensions is time-consuming and error-prone. Despite the existence of these parallel languages, there are some definite advantages to write programs in a well-understood sequential language such as Fortran, and then having them automatically transformed into parallel code.

The focus of research on parallelizing compilers has been on the automatic translation of sequential code into parallel code. In the past two decades, considerable effort has been devoted to the development of parallelizing compilers that can automatically exploit the parallelism in programs written in conventional imperative programming languages, particularly Fortran. Much progress has been made with parallelizing compilers. Some examples are Parafrase [96] at the University of Illinois, Urbana-Champaign, PFC [8] at Rice University, PTRAN [9] at the IBM T. J. Watson Research Center, and Cedar [97] at the University of Illinois, Urbana-Champaign.

The main advantages of using conventional languages with parallelizing compilers are the following:

- Programmers still use their familiar conventional programming languages, and

leave the unfamiliar jobs, – finding parallelism and specifying synchronization, – to compilers.

- It is not required that programmers understand the architectures of the target machines.
- Many existing programs can still run on parallel machines without modifications.

With sufficiently good parallelizing compilers, programmers could code in standard sequential languages and let the compiler worry about parallelism. Smart compilers, it is thought, can detect most of parallelism in a program automatically. The quality of the results depends on the structure of the algorithm and the architecture of the target machine. Good and thorough parallelization of a program depends critically on the degree to which a compiler can discover the data dependence information, which serves as a major source from which parallelizing compilers generate parallel code. Extensive research has been carried out concerning the automatic extraction of parallelism from sequential code. Efficient solutions have been found to many problems. Existing techniques that are used in optimizing sequential compilers, such as data-flow analysis, can be used in parallelizing compilers. With the advent of parallel computer architecture, these techniques, as well as new techniques [96, 97], provide some successful methods to support the automatic mapping of sequential programs to the new parallel architectures.

Important work on parallelizing compilers has been done to exploit the parallelism present in loops. There are several reasons that so much work has been concentrated on loop analysis:

- loops usually account for a large portion of scientific programs;
- loops require a large part of the execution time;
- loops often comprise a large amount of the parallelism.

Many algorithms have been developed to detect the dependencies among different iterations of a loop. In addition, a variety of different scheduling strategies can be used to determine which iterations should be executed by which processes.

The actual parallelism in a program is constrained by various dependencies. Dependences can be categorized into three types: resource, data, and control. To aid

the discussion of parallelizing compiler issues, we give some definitions for these dependencies. For simplicity, we omit functions/procedures in a program.

Definition 1.1.1 *Let P be a program, i.e., $P = \{s_i : s_i \text{ is a statement}, 1 \leq i \leq n\}$; denote by s, a statement of P, and v, a variable in P.*

(i) use(s) = {r: r is a limited physical resource used by s}

(ii) var(s) = {v: v is a variable occurring in s}

(iii) input(s) = {v: v ∈ var(s) & v is an input variable of s}

(iv) output(s)= {v: v ∈ var(s) & v is an output variable of s}

The above definition of $use(s)$ defines limited physical resources used in the statement s. These physical resources can be output ports, terminals, processors, etc. $var(s)$ defines the variable set used in the statement. For example, if a statement k is defined as

```
k: a = b + c;
```

then $var(k) = \{a, b, c\}$. The definition of $input(k)$ defines the input set of variables of the statement k; $output(k)$ defines the output set of variables of the statement k. If k is the statement above, then $input(k) = \{b, c\}$, $output(k) = \{a\}$.

Definitions for each type of dependence are in the following:

Definition 1.1.2 *Let s_i and s_j be two statements in a program P with $i \neq j$. If*

$$use(s_i) \cap use(s_j) \neq \emptyset$$

then s_i and s_j are resource-dependent.

Resource dependence between two statements is a consequence of the limited availability of hardware in any physical computer system. This type of dependence occurs when two different operations simultaneously attempt to use the same physical resource, such as when two multiple operations compete for an output port.

Definition 1.1.3 *If the execution of a statement s_i determines whether or not a statement s_j is to be executed, then s_j is control-dependent upon s_i.*

Intuitively, a control dependence from statement s_j to statement s_i exists when statement s_j should be executed only if statement s_i produces a certain value. This type of dependence occurs, for example, when s_i is a conditional statement and s_j is to be executed only if the condition evaluates true.

Definition 1.1.4 *Let s_i and s_j be two statements in a program P. If $v \in output(s_i) \cap v \in input(s_j) \neq \emptyset$, or $v \in output(s_i) \cap v \in output(s_j) \neq \emptyset$, or $v \in input(s_i) \cap v \in output(s_j) \neq \emptyset$, we say that s_j and s_i have data dependency.*

For example, in the following program:

s_1: `A = b + c;`

s_2: `d = A + e;`

s_2 needs the value of `A` produced by s_1 before it can begin its execution.

Hypothesis 1.1.1 *If there is no resource dependence, control dependence, or data dependence existing between two statements, then the two statements can be executed in parallel.*

The purpose of dependence analysis is to provide information for process scheduling. The independent threads which have been located during dependence analysis are to be scheduled to execute on processors. As we indicated before, many techniques developed for code optimization can be used for dependence analysis. However, the general problem of determining whether or not there is a dependence between two subscripted variables in a nested loop is undecidable. Effective tests can be developed only if we apply some restrictions. One question is: can we determine if there is a dependence between two statements in general?

Theorem 1.1.1 *It is undecidable in general whether two statements in a sequential program can be executed in parallel.*

Proof. Suppose we have the following program:

```
      ......
s1: i = f1();
s2: j = f2();
s3: k[i] = ...;
s4: k[j] = ...;
      ......
```

We assume that f_1 and f_2 are two arbitrary functions with no arguments and that there is no resource or control dependence between the statements s_3 and s_4. According to Hypothesis 1.1.1, statement s_3 and statement s_4 can be executed in parallel if and only if there is no data dependence between them.

Without loss of generality, we restrict $input(s_2) \cap output(s_1) = \emptyset$, and $input(s_4) \cap output(s_3) = \emptyset$. Then statement s_3 and statement s_4 can be executed in parallel if and only if $output(s_1) \cap output(s_2) = \emptyset$, i.e., $i \neq j$. It is well-known that for two arbitrary independent functions f_1 and f_2 without parameters, it is undecidable whether they return the same value. Therefore, we cannot decide if there is data dependence between statement s_3 and statement s_4. □

It is true that many kinds of dependence can be detected in programs. However, code generation for a target parallel machine can still be intractable. In particular, task scheduling needs to ensure that control dependencies, data dependencies, and resource limitations are properly handled during concurrent execution. The goal is to produce a schedule that minimizes the execution time or memory demand in addition to ensuring the correctness of the execution. The scheduling problem for parallel computing has received considerable attention in recent years. It is one of the most challenging problems in parallel processing, and it is known to be NP-complete in its general form [114].

Various scheduling techniques have been developed to improve the performance of computation. Even though the scheduling problem is NP-complete, many researchers have studied restricted forms of the problem. For example, when communication between tasks is not considered, a polynomial time algorithm can be found for scheduling tree-structured task graphs where all tasks execute in one time unit [71]. An optimal schedule determines both the allocation and the execution order of each task in such a way that all the tasks are completed in the shortest elapse time.

Since parallel architectures differ in synchronization overhead, scheduling constraints, memory latencies, and other implementation details, it is difficult to develop a general approach to exploit parallelism. Moreover, even if parallelizing compilers can detect all the hidden parallelism in a sequential algorithm, they cannot, in general, construct a new algorithm that is better suited to a specific parallel architecture. Many algorithms are inherently sequential. For example, it would be extremely difficult to convert a specialized sequential sorting algorithm into an efficient parallel algorithm automatically. These factors imply that restructuring is best tackled in a system where the user can make strategic decisions and supply the system with information that cannot be obtained automatically.

We have also observed that most parallelizing compilers have been developed for Fortran. This is because Fortran is the most popular programming language in the

scientific community which is the main user group that parallelizing compilers are intended to serve. As other programming languages, such as C and Pascal, become more popular, the difficulty of developing parallelizing compilers will increase, because the pointers used in C and Pascal make the analysis of data dependence much more difficult. The presence of unrestricted pointers in C reduces opportunities for automatic parallelization, because it is impossible to determine which variable may be referenced by a pointer. In spite of all of the above problems, parallelizing compilers are still very useful in exploiting a significant subset of parallelism hidden in a sequential program. However, it is clear that, in order to further exploit parallelism, new language constructs are needed, so that the programmers can use them to supply information on parallelism to the compiler.

Languages with Low-level Parallel Constructs

Many existing commercial parallel systems only support conventional programming languages with a few low-level constructs for process creation, termination, and synchronization. Many of the constructs developed take form of system calls. The major reasons for taking this approach are: (1) it is easy to implement; (2) there is no high-level parallel language construct that has been commonly accepted.

This approach typically requires changes in some or all of the following areas:

- support of reentrancy,
- creation and termination of parallel processes,
- synchronization of parallel processes, and
- distinguishing between process-private and process-shared data.

In many cases, the extension can consist solely of new library routines, leaving the original language and its compiler untouched. This makes it relatively easy and inexpensive to experiment with the various extensions.

In Sequent parallel C (DYNIX C), for example, a keyword **shared** allows the user to designate global data accessible by all processes. Other operations, such as process creation and termination and synchronization, have to be specified by library routines. The `m_fork` function creates a set of parallel processes to execute a function concurrently. The function `s_init_lock` initializes a memory-based lock. After the lock is

initialized, a shared variable can be locked with the `s_lock` routine and unlocked with the `s_unlock` routine. Sequent also provides general semaphores for synchronization; the function `semget` is used to get a semaphore, and the function `semctl` provides a variety of semaphore control operations. These semaphore functions can be very complicated to use.

In nCUBE-2 hypercube systems, a programmer has to use message-passing constructs to deal with synchronization and communications. Also, the programmer must write two separate programs for the host and the nodes. The host program performs terminal I/O, allocates the hypercubes, and loads the node programs. It serves as an interface between the user and the nodes. The node programs can run on specified nodes of the cube. Each node receives its initial data from the host or other nodes and sends its portion of the result back to the host. There are different sets of commands and built-in functions for the host and for the nodes.

Even though programming in languages with low-level parallel constructs can produce highly efficient code, the constructs are not compatible with other constructs of the language. Such architecture-dependent language extensions and library routines make the language unportable. Obviously, dealing with low-level language constructs or functions makes programming and debugging very difficult.

Languages with High-level Parallel Constructs

New languages or language constructs can be designed to allow users to express parallelism explicitly and conveniently. Concurrent Pascal [57], Path Pascal [26], Linda [28], Occam [113], Ada [108], and Jade [80] are examples. Programmers need more than only low-level mechanisms; high-level parallel language constructs can specify, in a way that is notationally convenient and conceptually elegant, both the events that can occur concurrently and the synchronization constraints between them. These high-level languages and language constructs for explicitly expressing parallelism and synchronization are essential in parallel programming.

As we stated above, parallel algorithms in general cannot be effectively implemented with programming languages that do not have facilities for the explicit specification of parallelism and a great deal of inherent parallelism cannot be derived automatically from sequential code. The explicit specification of parallelism, however, does not mean that programmers should be concerned with the details of a machine architecture. The programmers should only have to specify what is to be done (or can

be done) in parallel rather than how it is to be executed by specific processors. The determination of how parallelism is to be performed should be resolved by compilers rather than programmers. Parallel languages with high-level constructs, then, should present the programmer with a model of computation that can encompass a wide variety of architectures.

Three issues distinguish parallel programming from sequential programming in a shared-memory system:

- whether the management of memory is shared or not,
- the use of multiple processors, and
- synchronization and communication among the processors.

Problems with memory arise if one tries to specify a data structure as shared by all processes. Because the operating system does not know how the data structure is represented, it is hard to use any system calls to specify it. In distributed-memory systems, programmers have to write explicit code that can feed the data structure into a sequence of bytes on the sending end and reconstruct the original data structure on the receiving end. A language construct designed for data management can reduce the complexity.

Parallel programs execute pieces of their code in parallel on different processors. The goal is to make optimal use of the available processors; decisions regarding what can be run in parallel are of great importance. The first requirement for parallel programming support is, therefore, the ability to assign the execution of different parts of a program to different processors.

The processes of a parallel program need to cooperate while executing a parallel application. Processes sometimes have to exchange intermediate results and to synchronize their actions. We address this issue in detail in the next section.

As mentioned above, a parallel programming language allows the direct coding of parallel algorithms. An efficient utilization of parallel resources can be truly achieved only when programmers can express their applications explicitly in parallel code. Besides the advantages described above of using parallel languages, parallel programming languages also improve readability, portability, and static type-checking.

1.2 Synchronization in Parallel Processing

In the development of high-level language constructs for parallel processing, the design of synchronization primitives becomes one of the essential issues for parallel programming languages. In vector processors and *SIMD* computers, synchronization is generally an explicit part of the control flow and is executed as part of every instruction. In MIMD multiprocessors, synchronization must occur on demand. Hence, more sophisticated schemes are needed.

The need for synchronization is due to two different considerations: competition for resources and coordination between processes, arising from applications. The *Mutual exclusion* problem, for example, is due to competition between different processes for access to the shared resources of the system. Access is mutually exclusive if no two processes access a shared resource simultaneously. Consider what happens when two processes P_1 and P_2 concurrently execute the statement $i = i+1$, with the shared variable i having an original value of 5. Suppose both P_1 and P_2 read the value of i before either of them update it: P_1 reads i, P_2 also reads i; P_1 increases the value and writes it to i, P_2 also increases the value and writes it to i. Then the value will be 6 rather than 7. To prevent this, access to i should be mutually exclusive, so that the read, increment, and write operations perform indivisibly. It should be noted that not only should access to i be exclusive but also the execution of the whole statement $i = i + 1$. Synchronization serves the purpose of enforcing the correct sequencing of processes and of ensuring mutually exclusive access to shared resources.

One or more processes may need to wait until a set of variables is in a specific state before proceeding. Any process attempting an operation should be delayed until the state is reached as a result of the execution of other processes. Synchronization in this situation serves the purpose of enforcing the correct execution sequence. This is called conditional synchronization. In a producer-consumer problem, for example, the consumer process has to delay the execution if the shared queue is empty. On the other hand, the producer process must also delay the execution if the shared queue is full. Waiting for the queue to be non-empty for the consumer, or non-full, for the producer, is an example of conditional synchronization.

In message-passing programming environments, processes share channels that provide communication paths between processes. The process interaction is the exchange of messages between processes by way of communication channels. Channels are accessed by means of two kinds of primitives: send and receive. Processes send and

receive messages instead of reading and writing shared variables. Synchronization is accomplished because a message can be received only after it has been sent; these conditions constrain the order in which two events can occur. Since the major purpose of this thesis is to address synchronization in a shared-memory system, we will not go into detail on the problems and solutions of synchronization in distributed systems.

Synchronization issues have been addressed from the hardware design level to the programming languages level. A multiprocessor system usually contains various mechanisms to handle synchronization requirements at different levels. Usually, low-level primitives are implemented directly by hardware, while more complex mechanisms are implemented through software. Typical examples of synchronization primitives implemented by hardware include: test-and-set, increment-and-decrement, compare-and-swap, and fetch-and-add. Since our goal is to design high-level synchronization mechanisms based on a software implementation, synchronization primitives implemented by hardware will not be reviewed in this thesis. Although synchronization has been studied extensively, existing mechanisms and their pitfalls still need to be readdressed in the context of parallel programming language constructs.

In the next chapter, we will discuss various synchronization mechanisms and consider their advantages and problems.

Chapter 2

Synchronization Primitives

Synchronization issues first arose in concurrent programming for uniprocessor systems. As a first step towards solving the problem of critical sections in concurrent programming, Dijkstra [37, 38] introduced the concept of semaphores and the P and V operations. Semaphores are used to solve many kinds of synchronization problems in concurrent programming.

The next step was the notion of a critical region, introduced by Brinch Hansen [54, 55] and Hoare [65]. The idea is not to protect segments of statements but to protect variables. A variable is either shared by more than one process or not shared. The shortcoming of the critical region is that a single shared variable could appear anywhere throughout an entire program. This could make it very difficult to understand how that variable is being used in the program.

A module is used to encapsulate one or more variables, together with the procedures that act on them, within a single syntactic unit. The variables of the module can appear only within the procedures of the module. Combining the notions of the module and the critical region, Brinch Hansen [56] and Hoare [66] proposed the concept of the monitor. A monitor is a module in which each procedure is implicitly a critical region. It is the responsibility of the compiler to ensure the mutual exclusion of the monitor procedures. The concept of the monitor has been very popular and has been implemented in languages such as Concurrent Pascal and Mesa. Unfortunately, some flaws prevent its continued acceptance. First, the monitor unnecessarily restricts concurrent activity [75]. At times when a monitor procedure is not acting on the shared variables, a process could be active in the part of the procedure while another process was actively using the shared variables. Also, the shared variables of a monitor might naturally fall into two or more partitions, such that one process

could be acting on some of the shared variables while another process was acting on the other. Second, monitors cannot be defined nestedly in general. The nested monitor problem and has been studied thoroughly, and no satisfactory solution has been found. The problem of the nested monitor remains a blemish on the monitor concept.

Path expressions were first introduced by Campbell and Habermann [25]. Path expressions provide an elegant notation for expressing synchronization constraints. They take some of the burden of the implementation details of synchronization off the programmers. However, path expressions are poorly suited to solving conditional synchronization problems, even though the late version of predicate path expressions solves some of the problems. The base language structure must also be changed if path expressions are integrated. Furthermore, solutions to many well-known synchronization problems are complex and difficult to understand.

The synchronization primitives discussed above are primitives based on shared variables. The implementation of such primitives requires shared variables supported either by hardware or software. Synchronization primitives based on message passing began to be developed when networks and parallel systems with distributed memories appeared. From low-level **send** and **receive** constructs to high-level remote procedure calls, the rendezvous model, and one-to-many message passing, synchronization mechanisms have been developed to suit the nature of message-passing systems.

In this chapter, we survey various synchronization primitives from the point of view of programming languages. We discuss the advantages and disadvantages of these synchronization mechanisms. We pay special attention to two programming languages, Ada and C-Linda, and focus on how synchronization constraints can be specified by their language constructs.

2.1 Evaluation of Synchronization Mechanisms

Because of the lack of a commonly accepted theoretical model to describe various aspects of parallel programming, it is difficult to establish a set of theoretical criteria to use in comparing the synchronization mechanisms of parallel programming languages. In addition, the properties of a synchronization mechanism, such as ease of use, high-levelness, and expressive power, are vaguely defined. In this thesis, we do not expect to develop a complete method to compare and evaluate existing synchroniza-

tion mechanisms. Rather, we use a set of examples to illustrate the different aspects of synchronization mechanisms from the point of view of programming languages. In these examples, we focus on the following properties of each synchronization mechanism: expressive power, level of the language constructs, and ease of use.

2.1.1 Evaluation Criteria

We address three major aspects of a synchronization mechanism. First of all, a synchronization mechanism should be powerful enough to express any practical synchronization constraints. Second, language constructs for synchronization should be at a level compatible with other high-level language constructs. Last, any practical synchronization constraint should be easily expressed by the synchronization mechanisms.

Expressive Power. Synchronization serves two purposes: enforcing the correct order of executing processes and ensuring mutually exclusive access to certain shared data or resources. Language constructs for synchronization should be able to express all synchronization constraints. Expressive power is also demonstrated by proving that one synchronization mechansim can emulate another. An easy way to measure the expressive power of a synchronization mechanism is to examine how it can be used to solve some typical synchronization problems.

High-levelness. There is always a concern that a high-level language construct may lead to a loss of flexibility in expression. We believe that high-level language constructs can be developed to be compatible with other high-level constructs and still retain flexibility. Furthermore, users should not have to deal with the details of how synchronization is implemented. High-level synchronization constructs should relieve users from the burden of implementation, requiring them only to specify synchronization requirements.

Ease of Use. Language constructs for expressing synchronization constraints must provide a straightforward way to specify both event ordering and mutual exclusion. In parallel programming, the choice between fine-grain and coarse-grain decomposition is based on several factors, including process management overhead, communication costs, and the conceptual grain size of the parallelism inherent in the application. Synchronization constructs should be able to express constraints for both fine-grain and coarse-grain synchronization problems.

2.1.2 Synchronization Problems

To evaluate existing synchronization mechanisms, we study how these synchronization mechanisms can be used to deal with some typical synchronization problems. The simple producer-consumer problem is as an example requiring mutual exclusion and event ordering. The finite buffer problem is used as a resource allocation example. The dining philosophers problem, which was originally stated and solved by Dijkstra [37], is considered as a classical synchronization problem. It is an example of a large class of concurrency control problems. These problems are used to test nearly every newly proposed synchronization scheme.

Simple Producer-Consumer Problem

Two processes run concurrently. One process produces a value and stores the value at a shared variable which we call w, and the other process uses the variable for its computation. This process repeats indefinitely. The solution to this simple problem is correct if the following criteria are satisfied:

- The first process must produce a value w before the other process consumes the value of w
- The next value of w can only be generated after the previous value of w has been consumed.
- Mutual exclusion must be guaranteed to access w.

Finite Buffer Problem

A circular buffer of finite length, say size n, is shared by a number of processes. Processes write in a buffer location pointed by a variable *rear* and destructively read buffer contents, one at a time, from a buffer location pointed by *front*. The two points move circularly along the buffer. The constraints to be imposed are:

- No process should be allowed to read when the buffer is empty.
- No process should be permitted to write when the buffer is full.

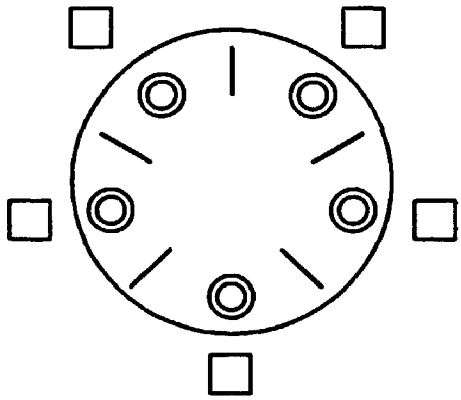


Figure 2.1: The Dining Philosophers Problem

The Dining Philosophers Problem

Five philosophers spend their lives thinking and eating. The philosophers share a round table surrounded by five chairs, each belonging to one philosopher. In the center of the table there is a bowl of rice, and five chopsticks are set on the table (see Figure 2.1). When a philosopher thinks, he does not interact with his colleagues. From time to time, a philosopher gets hungry and tries to pick up the two chopsticks that are closest to him, i.e., the chopsticks that are between him and his left and right neighbors. A philosopher may only pick up one chopstick at a time. When a hungry philosopher has both his chopsticks at the same time, he eats without releasing his chopsticks. When he has finished eating, he puts down both of his chopsticks and starts thinking again.

2.2 Semaphores

A semaphore is a nonnegative integer that can be accessed by two atomic operations:

`P(s)`: Delay the calling processes until $s > 0$ and then execute $s \leftarrow s - 1$

`V(s)`: Execute $s \leftarrow s + 1$

Semaphores that have only the possible values of 0 and 1 are called binary semaphores. Those that can take values from 0 to n are general or counting semaphores, which can be used when there are multiple identical instances of a shared resource.

Semaphores are used both for mutual exclusion and for conditional synchroniza-

tion. To provide mutual exclusion, a semaphore s is associated with a critical section and is used to guarantee sequential access to it. Before entering the critical section, each process must execute $P(s)$; upon exiting , it must execute $V(s)$. To perform conditional synchronization, a binary semaphore is associated with each condition: a process executes $V(s)$ on the semaphore when it sets the condition to true. Conditional synchronization can also be performed by associating a binary semaphore with each process. When a process is waiting for a condition, it performs an operation P on its private semaphore: a signaling process will perform a V when a condition becomes true.

First we will show how semaphores can be used to solve the producer-consumer problem:

```
semaphore full, empty, mutex ;
shared int w; /* shared variable */
 ......
init(mutex); /* mutex = 1 */
init(full);  /* full = 0 */
init(empty); /* empty = 1 */
/* process P1 */
while( ; ; ){
     P(empty);
     P(mutex);
     w = ...; /*produce w */
     V(mutex);
     V(full);
}
 ......
/* process P2 */
while( ; ; ){
     P(full);
     P(mutex);
     r = w + 1; /*consume w */
     V(mutex);
     V(empty);
}
```

.....

The finite buffer solution [109] using semaphores is as follows:

```
buffer;
defines put, get;
 const
    n = 100;(* capacity of buffer *)
 var
    buff: array 1..n of message; (* buffer *)
    inpt, outpt: 1..n;
    nbr_mess: semaphore init 0;
    nbr_free: semaphore init n;
procedure put (m: message);
code
    P(nbr_free);
    buff[inpt] := m; (* puts the message *)
    inpt := inpt rem n + 1;
    V(nbr_mess);
 end put;
procedure get (var m: message);
code
  P(nbr_mess);
  m := buff[outpt]; (* takes the message *)
  outpt := outpt rem n + 1;
  V(nbr_free);
end get;
code (* initialization *)
  inpt := 1; outpt:= 1;
end buffer;
```

The dining philosophers problem can be coded using semaphores as follows:

```
int J[4] = 0;
semaphore prism[4] = 0;

void test(int K){
```

```
    while ((C[(K-1)%5] != 2) && (C[K] == 1) &&
           (C[(K+1)%5] != 2)){
      C[K] = 2;
      V(prism[K]);
    }

main(){

semaphore mutex = 1;

    for(; ; ){     /* every philosopher */
       /* think */
       ......
       P(mutex)
       C[w] = 1;
       test(w);
       V(mutex);
       P(prism[w]);
       /* eat */
       ......
       P(mutex);
       C[w] = 0;
       test( (w+1) % 5);
       test( (w-1) % 5);
       V(mutex);
    }
}
```

Expressive power is an important criterion for a synchronization mechanism. Some observations have been made to determine whether semaphores can solve all possible synchronization problems [98, 78, 4, 61]. In [78], Kosaraju proves that certain synchronization problems cannot be realized by semaphores. However, semaphores are still considered to be one of the most powerful language constructs to deal with synchronization. Almost all practical synchronization requirements can be provided by semaphores.

The fundamental problem with using semaphores to specify synchronization is that they are too primitive to be compatible with other high-level language constructs. The responsibility of controlling executions remains with the programmer, who must decide when an execution can occur and on what conditions. Semaphores are rather a mechanism to enforce synchronization, but not a construct for the specification of synchronization.

Although semaphores can be used to program almost any kind of synchronization, `P` and `V` are rather unstructured primitives, so it is easy to make errors when using them. Semaphores provide no automatic enforcement of mutually exclusive access to shared variables. By mistake, a programmer may fail to include all references to a shared variable in critical sections. A forgotten `P` or `V` can quickly deadlock a program. And, since the synchronization code may be sprinkled throughout the program, unless the code is carefully structured, programs using semaphores are very difficult to understand, to maintain, and to prove correct.

2.3 Conditional Critical Region

The concept of a conditional critical region was introduced in [65, 55, 56]. The language construct, called a critical region, can be defined as follows:

A variable v of type T, which is to be shared among many processes, can be declared:

var v: **shared** T;

The variable v can be accessed only inside a *region* statement of the following form:

region v **do** S;

This construct means that while the statement S is being executed, no other process can access the variable v.

The critical region can be effectively used to solve the critical section problem. It cannot, however, be used to solve some general synchronization problems. For this reason, the *conditional critical region* was introduced. The major difference between the critical region and the conditional critical region constructs lies in the region statement; in the conditional critical region it has the form:

region v **when** B **do** S;

Here B is a Boolean expression. As before, regions referring to the same shared variable exclude each other in time. Now, however, when a process enters the critical section region, the Boolean expression B is evaluated. If the expression is true, statement S is executed. If it is false, the process relinquishes the mutual exclusion and is delayed until B becomes true and no other process is in the region associated with v.

Let us illustrate these concepts by coding the finite buffer problem [104]:

```
var buffer: shared record
                pool: array[0..n-1] of item;
                count, in, out: integer;
            end;

/* producer process */
region buffer when count < n
       do begin
          pool[in] := nextp;
          in := in + 1 mod n;
          count := count + 1;
       end;

/* consumer process */
region buffer when count > 0
       do begin
          nextc := pool[out];
          out := out + 1 mod n;
          count := count -1;
       end;
```

Programs written in terms of conditional critical regions can be understood by using an axiomatic approach. Each conditional critical region statement implements an operation on the resource that it names. Thus, once appropriate resource invariants have been defined, a concurrent program can be understood in terms of its component sequential processes.

The primary purpose of critical regions is to protect shared resources, in order to ensure mutual exclusion. Although conditional critical regions attempt to address

other aspects of synchronization requirements, i.e., event ordering, they led to more language constructs in order to meet the requirements. In addition, some weaknesses of the concept of conditional critical regions can be found in the following:

- The resource concept is unreliable: he same variable may be treated as a scheduled resource in some contexts and as an ordinary variable in other contexts. This may enable one process to refer directly to a variable while another process is within a critical region on the same variable.

- It is not natural to use conditional critical regions for process ordering. Dummy variables have to be used within conditional critical regions to enforce process ordering. This complicates the program.

- The context switching is inefficient: it does not seem possible to implement conditional critical regions efficiently. The problem is to limit the repeated evaluation of Boolean expressions until they become true.

- Conditional critical regions are not suitable for fine-grain synchronization problems. When requirements for synchronization have to be in a single statement level, using conditional critical regions becomes awkward.

Programs with conditional critical regions for synchronization can be difficult to comprehend. The conditional critical region statements that perform operations on resource variables are dispersed throughout the processes. This means that one has to study an entire concurrent program to see all the ways in which a resource is used. Every procedure which encounters a critical section has to provide its own synchronization explicitly.

Attempts to solve these problems eventually led to the development of monitors.

2.4 Monitors

A monitor is formed by encapsulating both a resource definition and the operations that manipulate it [37, 56, 66]. This allows concurrent access to a resource to be viewed as a module. Consequently, a programmer can ignore the implementation details of the resource when using it, and can ignore how it is used when programming the monitor that implements it.

A monitor consists of a collection of permanent variables that are used to store the resource's states and some procedures which implement operations on the resource. A monitor also contains permanent-variable initialization code, which is executed once before any procedure body is executed. The values of the permanent variables are retained between activations of monitor procedures and may be accessed only from within the monitor. Monitor procedures can have parameters and local variables, each of which takes on new values for each procedure activation. The structure of a monitor with name *mname* and procedures *op1*, ..., *opN* is shown in the following:

```
monitor
    var declarations of permanent variables;
    procedure op1(parameters);
      var declarations of variables local to op1;
      begin
          code to implement op1
      end;
    .....
    procedure opN(parameters):
      var declarations of variables local to opN;
      begin
          code to implement opN
      end;
    begin
      code to initialize permanent variables
    end
end monitor
```

Procedure *OpJ* within monitor *mname* is invoked by executing

call mname.*opJ*(arguments)

The invocation has the semantics usually associated with a procedure call, except that the execution of the procedures in a given monitor is guaranteed to be mutually exclusive. This ensures that the permanent variables are never accessed concurrently.

Various additional constructs have been proposed to realize conditional synchronization in monitors. For example, in [66], a condition variable is used to delay process

execution in a monitor; it may be declared only within a monitor. Two operations are defined on condition variables: **signal** and **wait**. If *cond* is a condition variable, then execution of *cond*.**wait** causes the invoker to be blocked on *cond* and to relinquish its mutually exclusive control of the monitor.

An example of a monitor that defines a finite buffer is given in the following.

```
type buffer(T) = monitor;

var slots: array [0..N-1] of T;
    head, tail: 0..N-1;
    size: 0..N;
    notfull, notempty: condition;

procedure deposit(var p: T);
    begin
      if size = N then notfull.wait;
      slots[tail] := p;
      size := size + 1;
      tail := (tail + 1) mod N;
      notempty.signal
    end;

procedure fetch(var it: T);
    begin
      if size = 0 then notempty.wait;
      it := slots[head];
      size := size - 1;
      head := (head +1) mod N;
      notfull.signal
    end;

begin
    size := 0; head := 0; tail := 0
end;
```

To deal with the difficulties in specifying conditional synchronization, the *condi-*

tional wait statement **wait** (B) was proposed [66]. B is a Boolean expression involving the permanent or local variables of the monitor. This synchronization facility is expensive, because it is necessary to evaluate B every time a process exits the monitor or becomes blocked at a conditional wait. An efficient variant of the conditional wait was proposed in [76] for use when only permanent variables appear in B. The finite buffer problem satisfies this requirement. Using this proposal, the finite buffer problem can be coded as follows [12]:

```
type buffer(T) = monitor;

var slots: array [0..N-1] of T;
    head, tail: 0..N-1;
    size: 0..N;
    notfull: condition size < N;
    notempty: condition size > 0;

procedure deposit(var p: T);
    begin
      notfull.wait;
      slots[tail] := p;
      size := size + 1;
      tail := (tail + 1) mod N;
    end;

procedure fetch(var it: T);
    begin
      notempty.wait;
      it := slots[head];
      size := size - 1;
      head := (head +1) mod N;
    end;

begin
    size := 0; head := 0; tail := 0
end;
```

The absence of a **signal** primitive is noteworthy. This approach provides an automatic signal, which is less error-prone than explicitly programmed **signal** operations. The primary limitation of the proposal, however, is that it cannot be used to solve most scheduling problems, because operation parameters that are not permanent variables, may not appear in conditions.

Other approaches have been proposed to deal with conditional synchronization, e.g., a primitive **notify** in Mesa [81].

Monitors can be used to implement various synchronization constraints. However, using monitors in a different way may lead to a serious problem known as the nested monitor call. When structuring a system of a hierarchical collection of monitors, it is likely that monitor procedures will be called from within other monitors. The nested monitor call problem occurs when a process, active in one monitor, enters another and then waits in that other monitor for some conditions to occur. While the process is waiting in the inner monitor, it holds exclusive rights to the original outer monitor. So no other process can enter the other monitor. In particular, this might include the process for which the original process is waiting in the inner monitor. This is the classic form of deadlock as each of two processes holds something for which the other is waiting. This problem has generated a lot of discussion [52, 86, 99, 120]. There are several unsatisfactory ways which have been proposed to deal with such nested monitor calls: (1) ignore them, as has been done in Concurrent Pascal, Pascal Plus [118] and Mesa; (2) prohibit nested monitor calls, as in Modula-1 [121], and as described in [74]; (3) acquire and release the mutual exclusion on all monitors along a call chain when a nested call is made and that process becomes blocked [87]; (4) release mutual exclusion on all monitors held by a process when the process waits on any condition [86, 120]; (5) release mutual exclusion on a monitor when a process leaves it to enter another monitor [52]. It is clear that each of the above approaches has flaws.

The monitor construct is a high-level language construct, and it takes certain implementation details of enforcing synchronization away from the users. The well-known programming languages which have implemented the use of monitors are Concurrent Pascal [57], Modula [121], and Mesa [81]. Synchronization using monitor operations is realized by code scattered throughout the monitor. However, some of the constructs, such as the **wait** and **signal** operations, are still visible to the programmer. Other code, such as the code ensuring the mutual exclusion of monitor

procedures, is not. Monitors have not taken away from the fine-grain mechanisms of semaphores. Support for monitor-based concurrency structure requires significant changes to a base language since monitors require a new envelope structure within the language, namely the monitor construct itself.

The concept of monitors can be easily understood by users. However, when using monitors to specify fine-grain synchronization constraints, such as at the single-statements level, the solutions can be unnecessarily complicated. In other words, monitors can be used to express coarse-grain synchronization constraints easily, but they do not appear to be suitable for fine-grain synchronization. In addition, when monitor constructs are integrated into a base language, the appearance of the language has to be changed. Programs with monitors become hard to read and understand.

2.5 Path Expressions

Path expressions were first defined by Campbell and Habermann [25]. Subsequent extensions and variations have also been proposed [82, 42, 83, 11, 62]. Here, we focus on one specific form of path expressions, open path expressions, which have been incorporated into Path Pascal [26].

The path expression mechanism permits synchronization to be specified by stating the set of allowed orderings of operations that access the resource. If a request is made for an operation on the resource, and that operation does not occur next in any sequence allowed by the path expression, then the process executing the operation is blocked until a state is reached in which that operation could occur next.

When path expressions are used, a module that implements a resource has a structure like that of a monitor. It contains permanent variables, which store the state of the resource, and procedures, which realize operations on the resource. Path expressions in the header of each resource define constraints on the order in which operations are executed.

The syntax of a path expression is:

path *path-list* **end**

A *path-list* contains operation names and *path operators.* Path operators include "," for concurrency, ";" for sequencing, "n : (*path-list*)" to specify up to n concurrent activations of *path-list*, and "[*path-list*]" to specify an unbounded number of concurrent activations of *path-list.*

If we have two operations *write1* and *write2* which try to access shared data, the mutual exclusion can be specified by the following path expression:

path 1: *write1, write2* **end**

The path expression:

path *write ; read* **end**

specifies that each *read* be preceded by a *write*; multiple activation of each operation can execute concurrently as long as the number of active or completed *read* operations never exceeds the number of completed *write* operations.

The path expression:

path 1: (*[read], write*) **end**

ensures that processes read consistent data; either an unbounded number of concurrent *reads* or a single *write* may be executed at any time.

The synchronization constraint for a bounded buffer of size N is specified by:

path *N: (1: (write); 1: (read))* **end**

1: (write) ensures that activations of the *write* operation are mutually exclusive, and *1: (read)* ensures that activations of the *read* operation are mutually exclusive. The operator ";" of *1: (write); 1: (read)* ensures that each activation of *read* is preceded by a completed operation *write*, and N restricts the number of completed operations of *write*, so that it is never more than the number of completed operations of *read*.

The bounded buffer can be defined as follows [13]:

```
module buffer(T);

    path N: (1: (write); 1: (read)) end;

    var slots: array [0..N-1] of T;
        head, tail: 0..N-1;

    procedure write(p: T);
        begin
```

```
            slots[tail]:=p;
            tail:=(tail + 1) mod N
        end;

    procedure read(var it: T);
        begin
            it:=slots[head];
            head:=(head + 1) mod N
        end;

    begin

        head:=0; tail:=0;
    end;
```

Let us look at the path expression solution for the Dining Philosophers Problem [27]:

```
const nphilosophers = 5;
      maxindex = 4;  (* nphilosophers - 1 *)
type diner = 0..maxindex;

var i: integer;
  table: object
    path maxindex:(starteating; stopeating) end
    var fork: array [diner] of
      object
        path 1:(pickup; putdown) end;
        entry procedure pickup; begin end;
        entry procedure putdown; begin end;
      end;

    entry procedure starteating(no: diner);
      begin
        fork[no].pickup;
        fork[(no+1) mod nphilosophers].pickup
      end;
```

```
    entry procedure stopeating(no: diner);
      begin
        fork[no].putdown;
        fork[(no+1) mod nphilosophers].putdown;
      end;
  end;   (* table *)

process philosopher(mynum: diner);
  begin
    repeat
      delay(ran(seed));
        table.starteating(mynum);
        delay(ran(seed));
        table.stopeating(mynum);
    until false;
  end;

begin
  for i := 0 to maxindex do philosopher(i)
end.
```

Let us look at the expressive power of path expressions. While path expressions provide an elegant notation for expressing synchronization constraints, they are poorly suited for specifying conditional synchronization [20]. Whether an operation can be executed might depend on the state of a resource in a way not directly related to the history of operations already performed. In fact, most resources that involve scheduling require access to parameters or to state information in order to make synchronization decisions. In order to specify solutions to such problems, additional mechanisms must be introduced. The need to realize conditional synchronization has motivated many of the proposed extensions to path expressions [11, 62, 42, 85]. Regrettably, none of these extensions has solved the entire problem in a way consistent with the elegance and simplicity of the original path expressions.

The path expression mechanism is very appealing for several reasons. First, path expressions are high-level language constructs. Path expressions were the first at-

tempt to express synchronization constraints rather than enforce them. They seem to take much of the burden of implementation away from the users. Second, they are designed specifically to be used as part of the definition of the abstract type of the resource. Synchronization is thus automatically associated with the resource.

Unfortunately, as we can see in the examples, path expression solutions to many standard synchronization problems are complex and difficult to understand. Using procedure names as the atomic elements of path expressions often causes unnecessary complexities in programs. For example, if a procedure is called from several different statements, the synchronization for these statements using path expressions requires the users to make several copies of the same procedure but with different names.

2.6 Synchronization Primitives Based on Message Passing

When message passing is used for communication and synchronization, processes send and receive messages instead of reading and writing shared variables. Synchronization is accomplished because a message can be received only after it has been sent, which constrains the order in which these two events can occur. Message passing is a different approach to synchronization. In message passing, all interprocess communication must take place through the explicit transmission of values between two or more processes. Traditionally, message passing has only been considered in the domain of distributed systems, where message passing is the sole means of interprocess communication. Today, several multiprocess systems are constructed based on message passing, such as nCUBE systems.

A message is sent by executing

send *expression-list*
to *destination-designator*

and a message is received by executing

receive *variable-list*
from *source-designator*

Designing message-passing primitives involves making choices about the form and semantics of these general commands. The implementation of **send** and **receive** has two primary parameters: how source and destination processes are named and how

communication is synchronized.

There are three possible methods for specifying the source and destination designator. The simplest method is direct naming, in which process names serve as source and destination. Some problems, such as the pipeline paradigm, which consists of n processes that communicate by the ith process, receiving messages from the $i-1$ process and sending messages to the $i+1$ process, are particularly well-suited for direct naming. However, this method is not well-suited to the kind of problem pose by client-server, where multiple servers serve any of multiple clients. The reason is that direct naming does not allow a process to receive messages from an arbitrary process except through the polling of all possible senders. A more general approach is global naming, or mailboxes. Instead of a process name, a mailbox is specified as the source or destination of a message. Any process associated with a mailbox can send to or receive from it. However, the cost may be quite high for all the members of a mailbox to maintain its current status. The third method combines direct and global naming. Any process can send a message to a port, but only one process can receive data from it. Ports provide the "anonymous" naming available via mailboxes without the additional overhead.

The other major design issue of message-passing systems is the choice between *synchronous* and *asynchronous* message passing. With synchronous message passing, the sender is blocked until the receiver has accepted the message explicitly or implicitly. Thus, not only do the sender and receiver exchange data but they are also synchronized. A message exchange always represents a synchronization point. With asynchronous message passing, the sender does not wait for the receiver to be ready to accept its message. Conceptually, the sender continues immediately after sending the message.

Asynchronous message passing implies unbounded buffer space. Because the sender never has to wait, a high degree of parallelism can be achieved. The drawbacks are that additional memory is required for buffering messages and that the information a process receives may be out of date in the asynchronous model. There are also some semantic difficulties to deal with. As the sender S does not wait for the receiver R to be ready, there may exist several pending messages sent by S, but not yet accepted by R. If the message-passing primitive is order-preserving, R will receive the messages in the order they were sent by S. The pending messages are buffered by the language run-time system or the operating system. The problem of a possible

buffer overflow can be dealt with in one of two ways. Message transfers can simply fail whenever there is no more buffer space. Unfortunately, this makes message passing less reliable. The second option is to use flow control, which means that the sender is blocked until the receiver accepts some messages. This introduces a synchronization between the sender and receiver and may result in unexpected deadlock.

In the synchronous message-passing systems there can be only one pending message from any process S to a process R. Buffering problems are less severe in the synchronous model, as a receiver needs at most one message from each sender, and additional flow control will not change the semantics of the primitive. On the other hand, the synchronous model also has its disadvantages. Most notably, synchronous message passing is less flexible than asynchronous message passing, because the sender always has to wait for the receiver to accept the message.

The **send** and **receive** primitives are sufficient to program any type of process interaction in message-passing systems. However, like semaphores, message passing is difficult to use correctly, because **send** and **receive** scattered throughout the program. In addition, such a point-to-point message establishes one-way communication between two processes. However many interactions between processes are essentially two-way in nature. For example, in the client-server model, the client requests a service from a server and then waits for the result to be returned by the server. This behavior can be simulated using two point-to-point messages. High-level language constructs have been proposed to provide direct support of synchronization and communication, and such constructs can be easier to use and more efficient to implement.

Remote Procedure Call

Remote procedure call (RPC) is another primitive for two-way communication. It resembles a normal procedure call, except that the caller and receiver are different processes. When a process S calls a remote procedure P of a process R, the input parameters of P, supplied by S, are sent to R. When R receives the invocation request, it executes the code of P and then passes any output parameters back to S. During the execution of P, S is blocked. S is reactived by the arrival of the output parameters. RPC is a fully synchronous interaction. When remote procedure calls are used, for example, the process S calls procedure *service* in the process R; the call statement in process S has a form similar to that used for a procedure call in a sequential language:

call *service (value-args; result-args)*;

The procedure *service* can be declared as a procedure in a sequential language.

remote procedure *service*
(**in** *value-parameter*; **out** *result-parameter*)
body
end

A major design choice is between a transparent and a nontransparent RPC mechanism. Transparent RPC offers semantics close to a normal procedure [19]. It gives the programmer a simple, familiar primitive for interprocess communication and synchronization. It also provides a sound basis for porting existing sequential software to distributed systems. Unfortunately, achieving exactly the same semantics for RPC as for normal procedures is close to impossible [112]. One source of problems is that, in the absence of a shared memory, pointers (address value) are meaningless on a remote processor. Because of the difficulty in achieving normal call semantics for remote calls, special features for remote calls have to be added. We can see such implementation in Modula-2 [10], a system can be considered as nontransparent RPC.

One-to-Many Message Passing

Many message-passing systems support a fast broadcast or multicast facility. A broadcast message is sent to all the processors in the system. A multicast message is sent to a specific subset of these processors. Broadcast and multicast are useful for operating system kernels and language run-time systems. For example, to locate a process providing a specific service, an enquiry message may be broadcast. Broadcast and multicast are also useful for implementing distributed algorithms, so some languages provide a one-to-many message-passing primitive.

One-to-many message passing has several advantages over point-to-point message passing. If a process needs to send data to many other processes, a single multicast will be faster and easier than many point-to-point messages. More importantly, from the synchronization point of view, a broadcast primitive may guarantee a synchronization point for all processes, or an ordering of messages that cannot be obtained easily with point-to-point messages [18].

2.7 Synchronization Constructs in Ada & C-Linda

2.7.1 Ada

Ada is the result of development efforts by the U.S. Department of Defense to define a standard programming language for embedded applications in which computer systems are part of larger physical systems such as communication systems. The result is a large, Pascal-like language that has capitalized on much of the programming language design research of the 70's. The design of Ada allows concurrent processes to execute in either shared memory or message passing systems.

Ada uses the rendezvous mechanism to enforce synchronization. For basic synchronization, Ada provides **entry** and **accept** statements. An **entry** specification has the same syntactic structure as the header of a procedure. Corresponding to an entry there exist one or more **accept** statements in the executable part of the task body. An **accept** statement executes when normal processing reaches it and another task executes a corresponding entry call statement. From the point of view of the calling task, both the syntax and semantics are that of a usual procedure reference.

Rendezvous

Two processes interact first by synchronizing, then exchanging information, and, finally, continuing their individual activities. This synchronization, or meeting, to exchange information is called a *rendezvous.* The rendezvous concept unifies the concepts of synchronization and communication between processes. The rendezvous model in Ada [115] is based on three concepts: entry declaration, entry call, and accept statement. The entry declaration and accept statement are part of the server code, while the entry call is on the client side. The accept statement can be of the following form:

accept *service (***in** *value-parameters;*
out *result-parameters)*
body
end

Syntactically, the entry declaration looks like a procedure declaration. An entry call

is similar to a procedure call statement.

One advantage of the rendezvous model is that client calls may be serviced at times of the server's choosing: for example, **accept** statements can be interleaved or nested. Another advantage is that the server can achieve different effects for calls to the same service by using more than one **accept** statement, each with a different body. The most important advantage is that the server can provide more than one kind of service. In particular, **accept** is often combined with selective communications to enable a service to wait for and select one of several requests to service. However, the rendezvous model also has problems similar to those of RPC, such as achieving exactly the same semantics as normal procedures.

With these language constructs, we can specify a simple producer-consumer problem. The two operations, *produce* and *consume*, are identified in the task specification. The task body repeats the **accept** statement pair for *produce* and *consume*:

```
task SimpleBuffer is
     entry produce(x: in message);
     entry consume(x: in message);
end;

task body SimpleBuffer is
     buffer: message;
begin
    loop
      accept produce(x: in message) do
          buffer:= x;
      end produce;

      accept consume(x: out message) do
          x:= buffer;
      end consume;
    end loop;
end SimpleBuffer;
```

Note that it is not possible for this task to service both the *produce* and *consume* operations simultaneously.

Both a call to an entry and the execution of the instruction **accept** constitute

requests for a rendezvous, and a task that makes such a request has to wait. The rendezvous takes place when the two partners, that is, the two tasks involved, arrive at the same entry simultaneously, in which case the rendezvous requests are queued, with a separate queue for each entry. However, **entry** and **accept** are not sufficient to solve some problems with greater complexity than that of the preceding example. For conditional synchronization, tasks need to respond to events in an unpredictable order. To allow for this capability, one of the three forms of a **select** statement can be used. The syntax of the **select** statement is as follows:

```
select_statement ::= selective_wait
                   | conditional_entry-call
                   | timed-entry_call
```

For example, the *selective_wait* statement provides a called task with a sophisticated means for selecting from a set of alternatives, each of which may depend on an associated guard.

```
selective_wait ::=
               select
                   select_alternative
               {or
                   select_alternative}
               [ else
                   {statement}]
               end select;

select_alternative ::=
    [ when condition = > ] selective_wait_alternative

selective_wait_alternative ::=
                 accept_alternative
               | delay_alternative
               | terminate_alternative

accept_alternative ::= accept_statement {statement}
```

```
delay_alternative :: = delay_statement {statement}

terminate_alternative ::= terminate;
```

With this construct, we can implement an Ada task that encapsulates a bounded buffer and the operations *read* and *write*:

```
task FiniteBuffer is
     entry writeBuffer(x: in message);
     entry ReadBuffer(x: out message);
end;

task body FiniteBuffer is
     N: constant := 0;
     buffer: array (0..N-1) of message;
     i, j: integer range 0..N-1 := 0;
     count: integer range 0..N := 0;
begin
     loop
       select
           when count < N =>
           accept writeBuffer(x: in message) do
               buffer(i) := x;
           end writeBuffer;
           i := (i + 1) mod N;
           count := count +1;
       or
           when count >0 =>
           accept ReadBuffer(x: out message) do
               x:= buffer(j);
           end ReadBuffer;
           j := (j +1) mod N;
           count := count -1;
       end select;
     end loop;
end FiniteBuffer;
```

It is unquestionable that rendezvous is a relatively high-level synchronization tool. However, as we have mentioned above, it has to be accompanied by a number of syntactic constructs in order to provide the desired flexibility. These include:

- the **select** statement
- guarded **select** statement
- **select** statement with branches **else**, **delay**
- attribute **count**.

With use of the rendezvous, we can often express synchronization in an attractive form. However, this may also generate a certain feeling of unease, arising from the fact that a task is made to play two distinct roles:

- the role of the task, and
- the role of a synchronization mechanism.

2.7.2 C-Linda

Linda, developed originally for the SBN network computer [46], consists of a few simple operations that embody the tuple-space model of parallel programming. Linda memory's storage unit is a logical tuple, or ordered set of values. While the elements of a conventional memory are accessed by address, elements in Linda memory have no addresses. They are accessed by logical name, where a tuple's name is any selection of its values. A Linda memory is accessed via three basic operations: read, add, and remove. Linda attempts to introduce language constructs for parallelism into a conventional programming language without disrupting the existing features of the language. Adding these tuple-space operations to a base language yields a parallel programming dialect. Linda operations have been added to C, Fortran, and other languages. Linda also runs on a wide range of parallel systems, i.e., shared-memory multiprocessors, distributed-memory multiprocessors and local area networks [28]. In recent years, Linda has become widely mentioned in the research on parallel and distributed programming languages.

Messages in Linda are never exchanged between two processes directly. Instead, a process with data to communicate adds it to the tuple space, and a process that needs

to receive data seeks it, likewise, in the tuple space. There are four operations over the tuple space: `out()`, `in()`, `read()`, and `eval()`. The operation out(t) causes tuple t to be added to the tuple space; the executing process continues immediately. The statement in(s) causes some tuple t that matches template s to be withdrawn from tuple space; the values of the tuple t are assigned to the formals in(s) and the executing process continues. If no matching t is available when in(s) executes, the executing process suspends until one is, then proceeds as before. If many matching t's are available, one is chosen arbitrarily. The operation read(s) is the same as in(s) with actuals assigned to formals as before, except that the matched tuple remains in the tuple space. The operation eval(t) is the same as out(t), except that eval adds an unevaluated tuple to tuple space [6].

The designers of Linda believed that "Forcing complex solutions to simple problems make us suspect that a language has chosen the wrong 'abstraction level' for its primitives, chosen operations with too many policy decisions built-in and too few left to the programmer. Such languages are ostensibly 'higher level' than ones with more flexible operations, but this kind of high levelness dissipates rapidly when programmers step outside the (often rather narrow) problem spectrum that the language designer had in mind" [28]. Linda indeed uses very primitive mechanisms for synchronization. We consider, however, that the problem is not with the level of synchronization constructs, but with the merits of the design of these language constructs. We can have high-level constructs for synchronization which are also very flexible to use.

The semaphore-like nature of `in()` and `out()` is one result of communication orthogonality in Linda. The operation in(*sem*) is functionally equivalent to the semaphore operation P(*sem*), and out(*sem*) to V(*sem*). More complicated synchronization mechanisms can be built using these operations. For example, to implement synchronization of message exchanges, in which the receiving operation suspends until a message is sent and the sending operation suspends until a message is received, Linda uses a macro facility:

```
def SYNCH_SEND(s: tuple)      [in(get); out(s); in(got)]
def SYNCH_RECV(s: tuple)      [out(get); in(s); out(got)]
```

To communicate synchronously, process A executes `SYNCH_SEND(B, message)` and process B execute `SYNCH_RECV(B, m:messge_type)`.

To conclude this chapter, we provide a C/Linda solution for the dining philosophers problem:

```
phil(i)
  int i;
{
  while(1) {
    think();
    in("room ticket");
    in("chopstick", i);
    in("chopstick", (i + 1) % Num);
    eat();
    out("chopstick", i);
  }
}

initialize()
{
  int i;
  for (i = 0; i < Num; i++) {
    out("chopstick", i);
    eval(phil(i));
    if (i < (Num - 1)) out("room ticket");
  }
}
```

Chapter 3

Synchronization Expressions(SEs)

3.1 ParC Overview

New constructs for synchronizations have been developed in the *ParC* parallel programming language project. These new constructs relieve programmers from the burden of imposing synchronization, requiring them only to specify the necessary constraints. Synchronization requests are specified by expressions of statement tags, termed synchronization expressions. The new constructs demand no structural changes to the base language and allow complicated synchronization constraints to be expressed easily. These new constructs can be naturally merged into its base language. In this chapter, we discuss the features of the new constructs, specify their syntax, and provide examples to show how they can be used.

The use of an established sequential language as a basis for the design of a new parallel language is an attractive approach, since it allows existing software to run on a new system without having to undergo major reprogramming. However, this approach can have the following drawbacks:

- The language mirrors the underlying seriality of the computers for which it was designed.
- Any changes introduced to the chosen language may appear to be patches on its original design. The changes may also lead to inconsistencies.
- The users, because of their familiarity with the sequential language, tend to design algorithms based on the sequential manipulation of data.

Sequential programming languages with low-level parallel constructs naively grafted onto them can be difficult to program. Parallelism should be integrated with the main structure of a programming language, and the new constructs should be able to help the programmer to cope with the extra complexity resulting from the introduction of parallelism.

An important aspect of a parallel programming language is the visibility of the target machine in the source language. If the target architecture is visible, the programmer may have the opportunity to choose a style of expression that is especially efficient for the target machine. Otherwise, the compiler or run-time system must be relied upon to perform such optimization where necessary. The latter approach is clearly more desirable, since it leads to more programs that are more machine-independent. Ideally, a parallel programming language should be independent in the same sense as the contemporary sequential languages, but, given the current state of the art, it may not be feasible to hide completely the gross structure of the target machines.

The motivation behind the design of our new language constructs is to relieve programmers from the burden of using low-level language constructs to specify parallelism and impose synchronization. Synchronization expressions is introduced to allow much more complicated synchronization constraints to be expressed and expressed easily.

ParC has the following features:

1. Shared variables have scopes.

2. Statements *PEXEC, PFOR*, and *SFOR* are provided for the explicit expression of parallelism.

3. Statement tags are introduced to identify the executions of statements.

4. Synchronization expressions (SEs), i.e., expressions of statement tags, are used to specify synchronization between processes.

The constructs in *ParC* for specifying parallelism are similar to those in other parallel languages. However, *ParC* takes a significantly different approach towards the specification of synchronization requirements. Statement tags and synchronization expressions are used for this purpose. Briefly speaking, a statement tag is a label in front of a statement that identifies the execution of the entire statement. Unlike

a goto label, it does not simply mark an entry point. A synchronization expression (SE) is an expression of statement tags, and it is used to specify synchronizations between processes. The purpose of introducing SEs is to allow synchronization to be specified directly, so that users need only specify *what* is to be synchronized rather than *how* to realize it.

3.2 Constructs for Parallelism

The first issue with which we must deal in a programming language for parallel processing is how to specify parallel execution. The nature of the issue becomes clear when we look at the traditional multiprogramming and concurrent programming. In an uniprocessor system, it is sometimes useful to express a program as a collection of processes running on a time-sharing basis; a given problem might lend itself well to being expressed as several independent processes, running logically in parallel, with at most one process running at a given moment in time. For example, the UNIX operating system provides several system calls to serve this purpose. This technique has been employed in programming languages, especially those intended for writing uniprocessor operating systems, for quite some time. We can learn a lot of experience to shift multiprogramming to parallel programming from the development of time-sharing systems.

Parallelism can be expressed in a variety of ways. An important factor is the unit of parallelism in a program. In a sequential program, the unit of parallelism is the whole program. In a parallel program, the unit of parallelism can be an object, a statement, an expression, or a clause (in logic languages). It ranges from fine grain to coarse grain.

In most imperative programming languages for parallel processing, parallelism is based on the notion of a process. Different languages have different definitions of this notion, but, in general, a process is a logical processor that executes code sequentially and has its own state and data. Processes are created either implicitly, by their declaration, or explicitly, by some specific constructs. For example, the `fork()` system call in UNIX creates a new process which carries the calling process description. For instance, both the child process and parent process have same file pointers.

With implicit creation, a programmer can also first declare a process type and

then create the processes by declaring variables of that type. Often, when we declare a queue for parallel processing, we declare a type of queue associated with processes applying operations on the queue. Once the queue has been used, all processes attached to this queue operate concurrently. In some languages, the total number of processes is fixed at compile time. This makes the efficient mapping of processes onto physical processors. However, it can also restrict the type of application which can be implemented with the language, as it requires that the number of processes be known in advance. Having an explicit construct for creating processes allows more flexibility. For example, the process-creation construct can allow parameters to be passed to the newly created process or processes, as with the `m_fork()` system calls in the DYNIX operating system of the Sequent. In the DYNIX, a `fork()` system call creates a new process, as in UNIX. The difference is that if there is a physical processor available, the `fork()` in DYNIX actually creates a copy of the calling process and assigns it to a physical processor. Otherwise, a virtual processor is created if there is no physical processor available. We can see two problems here with the `fork()` system call. First, the child process is basically a copy of the calling process, and it inherits almost all the attributes of the parent process. If the calling process has a large number of variables declared, the system call can be slow, because it needs to copy all the variables, and it can be wasteful, since the child process may need only a small portion of all the variables. Second, no parameters can be passed in the `fork()` system call. The system call can be difficult to use when programmers need to pass different parameters into different processes. In contrast, the `m_fork` routine in DYNIX assigns a subprogram to each child process, which then allows each child to run a different subprogram in parallel. In order to use `m_fork()` correctly, a number of other routines have to be used, such as `m_set_procs` to set the number of child processes which need to be created, and `m_getmyid` to find out the identification number of a child process, etc. We realize that such constructs for creating processes are low-level system calls which are machine-dependent and easily cause errors.

We have a look at how parallelism is expressed in Occam [72]. Occam allows consecutive statements to be executed either sequentially, as in:

```
SEQ
  statement_1
  statement_2
ENDSEQ
```

or in parallel, as in the following:

```
PAR
  statement_1
  statement_2
ENDPAR
```

This method is easy to use and understand. Initiation and termination of parallel computations are well defined. But the PAR construct does not support expressing SPMD-type parallelism with the number of processes unknown before execution. In such a case, a parallel loop statement can be used in Occam. It is similar to a traditional loop statement, except that all iterations of the loop can be executed in parallel, as in the following:

```
PAR i = 0 TO n
  a[i] := a[i] + 1
ENDPAR
```

Two principles are applied in our design of constructs for expressing parallelism:

- The programmer should be able to express explicitly and easily what could be done in parallel.
- The programmer should not be responsible for the chore of process management.

ParC has the following three statements for expressing parallelism:

1. *Parallel execution* statement

   pexec {
   statement_1;
   .....
   statement_n;
   }

   The n statements may be executed in parallel. No synchronization constraints are imposed by this construct itself.

2. *Parallel for* statement

   pfor ($expr\_1_{opt}$; $expr\_2_{opt}$; $expr\_3_{opt}$) *statement*;

   Each iteration of the loop is a separate process. The processes may be executed in parallel. No synchronization constraints are imposed by this construct itself.

3. *Synchronized for* statement

sfor ($expr\_1_{opt}$; $expr\_2_{opt}$; $expr\_3_{opt}$) {

statement_1;

.....

statement_n;

}

Each iteration of the loop is a separate process. The processes are to be executed in parallel. The ith statements for each $1 \leq i \leq n$, in all the processes, are to be synchronized (i.e., the so called lockstep synchronization).

In *ParC*, the expression of sequential execution follows the natural sequence of the program.

3.3 Shared Variables

In the previous section, we have discussed how to express parallelism in *ParC*. In this section, we discuss how shared data can be specified in *ParC*.

In *ParC*, a key word **shared** is used as a shared-memory storage specifier. It is similar to other C language storage specifiers, such as **auto**, **register**, **extern**, etc., which can be used before a type declaration of variables. If **shared** has been specified for a variable, the variable will be used as a shared datum.

```
shared int a;
shared struct token {
                int count;
                char *string;
                int flag;
                }
```

The variable **a** has been declared as a shared integer, **token** a shared structure.

ParC also supports the scope rule for shared variables. At present, most existing parallel languages support shared variables only at the global level. This restriction causes inconvenience in many cases. This can be illustrated by the following example:

```
pfor (i=1; i<=n; i++){
    shared int k;
    ......
    pexec{
            {
            ......
            k = k + local1;
            ......
            }
            {
            ......
            k = k + local2;
            ......
            }

          }
      ......
}
```

Here, a **pexec** statement is nested in a **pfor** statement. The variable k is intended to be a shared variable local to each process spawned by the **pfor** statement. In other words, there are n distinct variables k, each being a shared variable within the scope of one of the parallel iterations of the **pfor** statement. Without the scope rule of shared variables, programmers may have to resort to shared arrays or other means of dealing with such cases. Then the code would be obscure and need much more programming effort.

3.4 Statement Tags

In most of the conventional sequential languages, statements can have labels which are used in conjunction with the goto statements. With the advent of a structured programming approach, the labeling of statements is considered unnecessary and almost evil. However, in parallel programming languages, because of the multiple execution threads, statement labels can provide a useful means of specifying synchronization over parallel executions. The reasons for choosing statements as the entities

for specifying synchronization are:

- From the programmer's viewpoint, statements are the lowest-level entities in a program;
- A compound statement can also be considered as a statement that gives the programmer a wide range of choice, varying from fine grain to coarse grain, to specify synchronization;
- It is simple and straightforward to introduce the necessary synchronization at the statement level;
- The programmer need not work out extra encapsulation mechanisms for synchronization.

In *ParC*, any statement can be tagged. For example, in

```
P:: { .....
    (statement-lists)
    }
```

P is a statement tag for the compound statement. To retain the spirit of structured programming, statement tags should be declared in *ParC*, like variable declarations. Also, the syntax and scope rule for statement tags are similar to those of variables in C. Statement tags can be of either scalar or array types. For example, the following is a valid declaration:

```
tag p, q, r[3][2];
```

Tags of a array type are useful inside **pfor** (parallel for) statements. However, it is usually not recommended to tag a statement inside a **pfor** statement that has a large number of iterations.

Tagged statements can be nested. For instance, in

```
P:: { .....
    Q:: i++
    .....
    }
```

if tag Q is declared in the scope of the compound statement P, then the statement $i++$ can be referred to as Q within P, and cannot be referred to outside P. If Q is declared at the same level as P, or at a higher level, then the statement $i++$ can be referred to either as P.Q or simply as Q.

A few remarks are given in the following:

(i) Statement tags are different from statement labels (**goto** labels). A statement tag identifies a whole statement whereas a (**goto**) label marks only the beginning of the statement.

(ii) Statement tags differ from process identifiers in the following ways: (1) A tag may identify a statement which is shared by several distinct processes. (2) Tags can be used to represent fine-grain program segments, whereas process identifiers are usually used to mark relatively large segments of code.

(iii) Statement tags cannot be replaced by procedure identifiers. This can be illustrated with an example. Consider a procedure P in a reentrant code that is being called twice in statements Q1 and Q2 respectively. It is hard to distinguish between the two calls by the procedure identifier but can be distinguished if tags are used. Further, the idea of using procedure identifiers for synchronization purposes is much more expensive to implement and often conflicts with the original advantages of using procedures.

We list some of the advantages on the choice of statement tags as the basic elements of synchronization expressions:

- **Flexibility.** Statements, compound or simple, can vary in grain size. This gives the much needed flexibility in handling various synchronization problems.
- **Consistency.** The basic units in synchronization are processes, which coincide with the execution of statements in imperative language. Thus, the use of statement tags is consistent with the functionality of synchronization expressions.
- **Convenience.** There is no need for the programmers to work out extra encapsulation mechanisms for the processes involved in synchronization. There are no structural changes to the base language for implementing synchronization expressions. Therefore, it is convenient for the programmers to write and modify their synchronization requirements.

If a statement is in a segment of program that is shared by several processes, we say this statement has multiple copies. Again, each copy of a statement may be executed many times. We call each execution an instance of the statement.

In the following program, we show how the synchronization constraints for the simple producer-consumer problem can be specified with a synchronization expression. In the program below, p is a statement tag for the producer and q for the consumer. The synchronization expression (p->q)*, which denotes p -> q -> p -> ..., specifies that q cannot start until p terminates, and the next p cannot start until the previous q terminates, and so on.

```
tag p, q;
restrict (p->q)*;
shared int B;
.............
pexec {
    {
     /* a loop */
     .............
     p:: B = ...;
     .............
    }
    {
     /* a loop */
     .............
     q:: A = B*2 + 5;
     .............
    }
}
```

3.5 Synchronization Expressions

In this section, we specify the syntax and semantics of our synchronization expression (SEs) and discuss the major differences between our SEs and path expressions (PEs). But first, we provide reasons why we need to develop synchronization expressions:

- Constructs for synchronization at the programming language level should be used to specify what the synchronization requirements are rather than how those requirements are to be implemented or imposed. However, most of the currently

used language constructs for synchronization are suitable only for imposing synchronization, not for expressing synchronization requirements. They are too primitive to be compatible with other constructs of high-level languages.

- The processes involved in synchronization in a parallel program range from fine-grain to coarse-grain. Thus, language constructs for synchronization should be flexible enough to deal with both fine-grain and coarse-grain processes. Procedure-based synchronization constructs are appropriate only for dealing with processes of medium to coarse grain. They are inflexible and inconvenient when dealing with fine-grain processes and complicated synchronization problems.

- Synchronization expressions are constructs for specifying synchronization requirements in a program. They relieve the programmer from the burden of imposing synchronization constraints, leaving the implementation issues to the compiler. They demand no structural changes to the base language, and they are easy and convenient to use for both fine-grain and coarse-grain synchronization problems.

3.5.1 Syntax of Synchronization Expressions

```
sync_spec   ::= RESTRICT sync_expr ';'

sync_expr   ::= /* empty */
              | sync_expr_1
              | sync_expr_2

sync_expr_1 ::= statement_tag '[]' statement_tag
              | sync_expr_1 ',' statement_tag '␣]'
                statement_tag
              | '(' sync_expr_1 ')' '*'
              | '(' sync_expr_1 ')' '*' WITH guard
              | '(' sync_expr_1 ')'
              | index_range ':' sync_expr_1

sync_expr_2 ::= statement_tag
```

```
| sync_expr_2 '->' statement_tag
| sync_expr_2 '|'  statement_tag
| sync_expr_2 '||' statement_tag
| sync_expr_2 '&'  statement_tag
| '(' sync_expr_2 ')' '*'
| '(' sync_expr_2 ')' '*' WITH guard
| '(' sync_expr_2 ')'
| index_range ':' sync_expr_2
```

In the above definition, **sync_expr_1** expresses exclusion relations between processes, while **sync_expr_2** specifies execution dependencies, e.g., sequencing, barrier, etc. The **guards** appearing in synchronization expressions are used to express the restrictions on the number of instances of certain tagged statements to be executed. For example, the guard `#p - #q in 0..4` imposes that, at any moment, the difference between the numbers of appearances of statements p and q should be between 0 and 4. The **index_range** is used when some statement tags in the expression are indexed by variables rather than constants. For example, `i=1..5: (P[i] || Q[i % 5 + 1])*` is equivalent to five synchronization expressions, one for each of the five values of `i`.

Declarations of SEs follow the syntax of variable declarations of C. SEs also obey the scope rules of C. We describe the semantics of SEs informally. To avoid abundant usage of the phrase "statements whose tags appear in the synchronization expression" in the sequel, "statements" and "statement tags" are used interchangeably, wherever there is no confusion.

3.5.2 Descriptions of Synchronization Expressions

Synchronization expressions impose certain constraints on the execution of the statements whose tags appear in the corresponding SE. An instance of a statement can be executed safely only if the execution does not violate the restrictions specified by the synchronization expression. If commencing the execution of a statement at the current state would cause a violation of a synchronization specification in a program, then the execution is delayed until it (the statement) can be executed safely. Various cases of restrictions are considered in the following. Since the purpose of this chapter is to explain the concepts rather than to give a formal specification, the following is

not intended to be a detailed definition. A formal definition using synchronization languages will be provided in Chapter 4. In the following, the instant of a statement means one instance of the execution of a statement. For example, a statement within a loop can involve many instances of the execution.

Exclusion The exclusive execution of statements P1 or P2 can be expressed as $P1$ [] $P2$. This synchronization expression restricts in that only one instance of the two statements is to be executed.

Sequencing A strict sequencing of two statements can be specified by infixing the operator "–>" between the statement tags. For example, $P1- > P2$ specifies that P2 is executed only after the completion of P1.

Join The join expression $P1 \parallel P2$ specifies that the execution of the processes denoted by synchronization expressions $P1$ and $P2$ can proceed in any order, possibly concurrently, and that the termination (of the execution) of both statements marks the completion of the synchronization expression.

Selection The selection expression P1 | P2 specifies that either P1 or P2 but not both can start an execution, and the termination of this execution marks the completion of the expression.

Repetition The expression P* means that the execution specified by the P is repeated zero or more times (in sequencing).

Intersection The expression P1 & P2 specifies that the execution of the processes must satisfy both expression P1 and P2.

Synchronization expressions control interaction among processes by defining the set of valid event orderings. Consider a scenario in which there are a producer and two consumers with a one-unit buffer. Either of the consumers may proceed only after the producer has produced something. If we tag the producer process as P and one consumer process as C_A, and the other as C_B, then this control can be guaranteed by the synchronization expression:

```
restrict P->(C_A | C_B)
```

If the above restriction is to apply indefinitely, we can simply write:

```
restrict (P->(C_A | C_B))*
```

This means continuous apply of the restriction as:

```
P -> (C_A | C_B) ->
P -> (C_A | C_B) ->
P -> (C_A | C_B) ->
......
```

Example 3.5.1 *Assume that there are three parallel processes and it is required that statement* r *in one process can start its execution only after both statements* p *and* q *in other processes, respectively, have terminated. Furthermore, there is no synchronization constraint between p and q. Then this synchronization requirement can be specified as*

```
(p || q) -> r
```

Example 3.5.2 *Statements p and q, as well as s and t, have the same synchronization relation as in the write/read problem. In addition, q and t also have the same relation, and there is no explicit synchronization constraint between p and s. These conditions can be given by the expression*

```
((p->q)* || (s->t)*) & ((q->t)* || p* || s*)
```

In addition to the restrictions imposed by the synchronization expression, the guard, if present, further constrains the execution of the statements. Such guards are meaningful only in the repetition construct; hence, they are allowed to appear only in conjunction with this construct. If a guard appears in conjunction with the repetition constructs, as `(P)* with #P in 0..n`, then the number of repetitions is restricted to, at most, *n*.

The following *ParC* solution to the finite buffer problem shows how a guard construct is used in practice:

```
main(){
  ......
  shared buffer_type buffer[n];
  tag      W, R;
  restrict (W [] R)* with #W-#R in 0..n;
  pexec {
       ......
```

```
        W:: {rear = ++rear % n;
             buffer[rear] = value
            }
        ......
        R:: {front = ++front % n;
             value = buffer[front]
            }
        ......
    }
```

We can also present a simple solution to the dining philosophers problem:

```
main()
{
    tag      E[5];
    restrict i=0..4: (E[i] [] E[(i+1)%5])*;
    shared int chopstick[5];
    int      i;

    pfor (i=0; i<5; i++) {
       while (1) {
          E[i]::{pickup(chopstick[i]);
                 pickup(chopstick[(i+1) % 5]);
                 eat();
                 putdown(chopstick[i]);
                 putdown(chopstick[(i+1) % 5]);
                }
       }
    }
}
```

Note that the synchronization expression

```
restrict i=0..4: (E[i] [] E[(i+1)%5])*
```

actually denotes the following five expressions:

```
(E[0] [] E[1])*, ..., (E[4] [] E[0])*.
```

They specify that no two neighboring philosophers can eat at the same time.

The above solution is much simpler and more comprehensible (for the programmer) than the one presented in [27]. It can be observed that our solution does not require additional processes, such as *Fork* and *Room* in [67], and it does not require the programmer either to code the busy status of the forks or to restrict the number of philosophers inside the room. These are significant improvements in the simplicity of programming. More examples can be found in the next section.

As we have mentioned before, a major difference between synchronization expressions (SEs) and path expressions (PEs) is that SEs use statement tags as their atomic elements rather than the procedure ids used by PEs. Both PEs and SEs are designed for specifying synchronization constraints on processes which may be executed in parallel. A process is fundamentally different from a procedure. Therefore, SEs which use statement tags are consistent with the very function of synchronization, but PEs which use procedure ids are not. Procedures are capsulations of sequences of instructions. They may or may not coincide with processes. However, a procedure is not a process, but a procedure call is. For example, a procedure p may be called in several different segments of a program which may be executable in parallel. Those different calls of p are different processes, which may require different synchronization constraints. PEs, which use procedures as their basic components, cannot express those constraints directly. For example, a procedure p is called by two parallel processes, and there is a certain synchronization constraint between the two calls. PEs cannot specify this constraint unless two extra procedures are defined for the two calls. PEs can be used conveniently for resource management, which is a type of data-dependency problem. But, they are unnatural for operation-dependency problems and even for certain types of data-dependency problems.

Synchronization between the two processes that are at the different iterations of a parallel loop belongs to a class of synchronization problems, which are often seen in scientific computing. Path expressions cannot be used or at least are very difficult to use, to express this class of problems, due to the fundamental structure of (procedure-like) path building blocks.

In [20, 13], it was mentioned that there is no way to use parameter values in PEs; PEs are poorly suited to specifying conditional synchronization. Predicate PEs [11] and open predicate PEs [62] were intended to solve these problems. But these extensions to PEs became too complicated to use and have not solved the fundamental

structure problem mentioned above. In SEs, tags are associated with statements including procedure calls. A tagged statement can be nested in a conditional statement. So the above mentioned problems can be easily solved by using our SEs together with the control flow of a program.

After examining various forms of path expressions, including original path expressions [25], open path expressions [26], predicate path expressions [11] and open predicate path expressions [62], we conclude that our synchronization expressions have not only kept the elegance of path expressions, but also overcome some of their drawbacks.

We summarize the advantages of SEs over existing synchronization mechanisms to conclude this chapter.

- Synchronization expressions are flexible enough to specify synchronization constraints ranging from fine-grain to coarse-grain levels. Because functions or procedures serve as the basic synchronization units, critical sections, monitors, and path expressions are considered as coarse-grain synchronization mechanisms. They are not suitable for specifying fine-grain synchronization constraints and their solutions to fine-grain synchronization problems can be unnecessarily complicated. By using statement tags, synchronization expressions can specify fine-grain and coarse-grain synchronization requirements in a unified term.

- Having tagged statements as the basic units in synchronization expressions is consistent with the concept of processes in parallel programming. Critical sections, monitors, and path expressions all take functions or procedures as their basic synchronization units. For example, if a procedure is called more than twice by different processes, the procedure has to be cloned with different procedure names in order to be synchronized using such synchronization mechanisms. The basic units in parallel programming that need to be synchronized are processes, not functions or procedures. Using functions or procedures as basic synchronization units conflicts with the concept of processes. To be consistent with the concept of processes, function or procedure calls have to be used instead of function or procedure names. Therefore, tagging statements which call the procedure or the function provides a straightforward way to maintain consistence with the concept of processes.

- Synchronization expressions can be well integrated into the base language. When critical sections, monitors, and path expressions are integrated into a base language, the appearance of the language has to be changed. The synchronization mechanisms cannot be considered as a natural part of the base language. This is one of reasons that solutions to many synchronization problems are hard to understand. However, synchronization expressions demand no structural changes to the base language, and are easy to use.

Chapter 4

Synchronization Languages(SL)

The necessity of a formal semantic description for SEs is apparent both in theory and in practice. For example, we should be able to test whether the two SEs

```
(a || (b -> c));
(((a || b) -> c) | (b -> (a || c)))
```

are equivalent. In practice, compilers should be able to check, for example, whether two SEs contradict each other. Compilers should also be able to do simplification on SEs. All these tasks rely on a formal semantic definition of SEs that models our intuitive and informal description of SEs. Besides, this model is expected to provide a systematic way for implementing synchronization controls specified by SEs.

In this chapter, we introduce a new family of languages named *synchronization languages*, which we use to give a precise semantic description of SEs. Under this description, relations such as equivalence and inclusion between SEs can be easily understood and tested. In practice, it also provides us with a systematic approach for the implementation as well as the simplification of SEs in parallel programming languages. In addition to its practical significance in parallel processing, the family of synchronization languages i itself ans interesting topic in formal language theory. We show that it is a proper subset of the intersection of the families of regular and start-termination languages (st-languages), where an st-language is a subset of the shuffle of the languages $(a_1a_{1t})^*$, ..., $(a_na_{nt})^*$. We show that each synchronization language is closed under the following rewriting rules: (1) $a_sb_s \rightarrow b_sa_s$, (2) $a_tb_t \rightarrow b_ta_t$, (3) $a_tb_t \rightarrow b_ta_s$, (4) $a_ta_sb_tb_s \rightarrow h_tb_sa_ta_s$, and also $h(a_ta_sb_tb_s) \rightarrow h(b_tb_sa_ta_s)$, for any morphism h that satisfies certain conditions which will be specified later in the thesis. We conjecture that the four conditions above are also sufficient for a

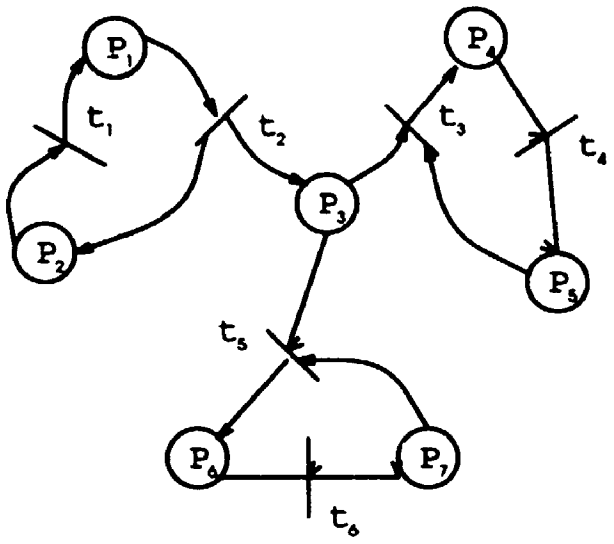


Figure 4.1: A simple graph representation of a Petri net

regular st-language to be a synchronization language. Several other properties of synchronization languages are also studied.

4.1 Theoretical Models for Parallelism

Many theoretical models have been developed to describe the behavior of concurrent processes. There are two basic types of such models: transition-based models and algebra-based models.

In transition-based models, Petri's net theory has been extensively studied [103, 102, 93, 78]. A Petri net is an abstract, formal model of information flow. The properties, concepts, and techniques of Petri nets are being developed in the search for natural, simple, and powerful methods for describing and analyzing the flow of information and control in systems, particularly systems that may exhibit asynchronous and concurrent activities. The primary use of Petri nets has been the modeling of systems of events that can possibly occur concurrently.

Figure 4.1 shows a simple Petri net. The Petri net graph models the static properties of a system. The graph contains two types of nodes: circles (called places) and bars (called transitions). These nodes, places and transitions, are connected by directed arcs from places to transitions and from transitions to places. If an arc is directed from node i to node j, then i is an input to j, and j is an output of i. In Figure 4.1, place P_1 is an input to transition t_2, while places P_2 and P_3 are outputs of

transition t_2. In addition to the static properties represented by the graph, a Petri net has dynamic properties that result from its execution. The execution of a Petri net is controlled by the position and movement of markers, called tokens, in the Petri net. Tokens reside in the places of the net. Although the Petri net model is very simple, it has been approached and utilized in a large number of ways, such as associating Petri nets with linear algebra, etc.

In 1973, Rabin showed that for two Petri nets C_1 with marking μ_1 and C_2 with marking μ_2, it is undecidable whether $R(C_1, \mu_1) \subseteq R(C_2, \mu_2)$, where $R(C_i, \mu_i)$ is the reachability set of Petri nets [103]. Furthermore, Hack later showed that it is undecidable whether $R(C_1, \mu_1) = R(C_2, \mu_2)$ [103]. These results and related undecidability results for Petri nets rule out the possibility of using Petri nets to model SEs because from the point of view of implementation it is essential to be able to decide whether given SEs are equivalent. In addition, SEs are only synchronization constraints; they are not the descriptions of all behaviors of a system. Hence, it cannot provide sufficient information for modeling using Petri nets.

The algebra-based models can be seen in the development of the calculus of communicating systems [90, 91, 92] and communicating sequential processes [22, 67], as well as the algebraic theory of processes [63], process algebra [17, 15], and trace languages [1, 89]. They provide a useful mathematical framework for the analysis of programs. However, these models do not provide a solution for the implementation of a synchronization control that allows two actions to overlap in time. In general, they consider concurrency as a nondeterministic sequential relation. Unlike transition-based models, algebra-based models do not take parallelism as a fundamental requirement. These models yield an inadequate description of concurrency. For example, CCS (calculus of communicating systems) expresses concurrency by sequential non-determinism: $(a \parallel b) = ab + ba$. Thus, the parallel composition operator can be eliminated.

Consider the simple SE $((p \parallel q) \rightarrow r)$ of Example 3.5.1, assuming that p, q, and r are all statement tags. Note that each statement tag identifies an execution of a statement whose duration in time cannot be ignored in general. Consider the trace language $\{pqr, qpr\}$. If we interpret concatenation in general as a sequencing relation, that is, pq means that q can start only after p terminates, then we lose the concurrency between p and q. If we consider concatenation as a sequence relation for only the starting points of executions, then we cannot express that r can start only

after both p and q terminate.

Attempts have been made to unify algebra-based models and transition-based models; see, e.g., [21], [94], and [105]. In [105], synchronized automata were introduced to model parallel threads. A synchronized automaton consists of two levels: a set of weakly synchronized automata and a set of states connecting them. The main problem with synchronized automata is that the states of weakly synchronized automata are not consistent with the states connecting them. In other words, there are two classes of states. This seems to be unnecessary and difficult to implement. The models introduced in [21] and [94] are also clearly not suitable for implementing synchronization expressions.

4.2 Basic Notation and Definitions

In this section, we give the basic notations and definitions for defining synchronization languages.

The family of subsets of a finite set A is denoted $\wp(A)$ and the cardinality of A is $\#(A)$. For $n \geq 1$, A^n is the cartesian product of n copies of the set A. For $x \in A^n$, x_i denotes the ith component of x, $i = 1, \ldots, n$.

Let Σ be a finite alphabet. The set of words (resp. nonempty words) over Σ is denoted Σ^* (resp. Σ^+). The empty word is denoted by λ. Subsets of Σ^* are called (Σ-)languages. A singleton language $\{w\}$, $w \in \Sigma^*$, is usually denoted simply as w. The set of subwords of $w \in \Sigma^*$ is denoted $\mathrm{sub}(w)$. The length of $w \in \Sigma^*$ is $|w|$. Let $L_1, L_2 \subseteq \Sigma^*$. The concatenation of L_1 and L_2 is $L_1 \cdot L_2 = \{w \in \Sigma^* | \exists u_i \in L_i,\ i = 1,2\ w = u_1 u_2\}$. For $L \subseteq \Sigma^*$, L^n denotes the concatenation of n copies of L and the Kleene *-operation is defined by $L^* = \bigcup_{i=0}^{\infty} L^i$. (Here $L^0 = \{\lambda\}$.) Also, $L^+ = L^* - \{\lambda\}$. The family of regular languages (over an alphabet Σ) is denoted REG (REG(Σ)). REG(Σ) is the smallest family of Σ-languages containing Σ and closed under (finite) union, concatenation, and the Kleene *-operation. The set of regular expressions over an alphabet Σ (and using the operations union $+$, concatenation $\cdot$, Kleene-closure *) is denoted reg(Σ) and the language represented by a regular expression $\alpha \in \mathrm{reg}(\Sigma)$ is $L(\alpha)$. Thus $\mathrm{REG}(\Sigma) = \{L(\alpha) | \alpha \in \mathrm{reg}(\Sigma)\}$. When there is no danger of confusion, we sometimes denote the language $L(\alpha)$ simply by the regular expression α.

Let $w = b_1 \cdots b_m$, $m \geq 1$, $b_i \in \Sigma$, $i = 1, \ldots, m$. Then let $\mathrm{first}(w) = b_1$ and $\mathrm{last}(w) = b_m$. The *shuffle* of words $u, v \in \Sigma^*$, $\omega_{sh}(u, v) \subseteq \Sigma^*$, consists of all words that

can be written in the form $u_1v_1 \cdots u_mv_m$, $m \geq 0$, where $u = u_1 \cdots u_m$, $v = v_1 \cdots v_m$, $u_i, v_i \in \Sigma^*$, $i = 1, \ldots, m$. The shuffle operation is extended to languages $L_1, L_2 \subseteq \Sigma^*$ in the natural way:

$$\omega_{sh}(L_1, L_2) = \bigcup_{w_i \in L_i, i=1,2} \omega_{sh}(w_1, w_2).$$

A *Thue system* is a finite set of rules $R \subseteq \Sigma^* \times \Sigma^*$. Elements of R are usually denoted as productions $w_1 \rightarrow w_2$, $w_1, w_2 \in \Sigma^*$. The productions of R determine the rewrite relation $\rightarrow_R \subseteq \Sigma^* \times \Sigma^*$ defined as follows. For $w_1, w_2 \in \Sigma^*$, $w_1 \rightarrow_R w_2$ iff there exists a production $r_1 \rightarrow r_2 \in R$ and $u, v \in \Sigma^*$ such that $w_i = ur_iv$, $i = 1, 2$. As usual, $\rightarrow_R^*$ denotes the reflexive and transitive closure of $\rightarrow_R$.

Let $L \subseteq \Sigma^*$ be an arbitrary language. We define

$$\Delta_R(L) = \{w \in \Sigma^* | (\exists u \in L) u \rightarrow_R w\}, \text{ and } \Delta_R^*(L) = \{w \in \Sigma^* | (\exists u \in L) u \rightarrow_R^* w\}.$$

We say that L is *closed under* R if $\Delta_R(L) \subseteq L$. The language $\Delta_R^*(L)$ is called the R-closure of L.

4.3 Synchronization Languages

Let Σ be a finite alphabet. The set of synchronization expressions over Σ,

$$\mathrm{SE}(\Sigma) \subseteq (\Sigma \cup \{\hat{\lambda}, \rightarrow, \&, |, \|, *, (,)\})^*,$$

is defined in the following. For $e \in \mathrm{SE}(\Sigma)$, we denote by $\mathrm{sym}(e)$ the set of symbols of Σ appearing in e. Intuitively, $\mathrm{sym}(e)$ is the minimal alphabet for the expression e. The set of *subexpressions* of e is denoted $\mathrm{sub}(e)$.

(i) $\Sigma \cup \{\hat{\lambda}\} \subseteq \mathrm{SE}(\Sigma)$.

(ii) Let $e_1, e_2 \in \mathrm{SE}(\Sigma)$. Then $e = (e_1 \rightarrow e_2)$, $(e_1 \mid e_2)$, $(e_1 \,\&\, e_2) \in \mathrm{SE}(\Sigma)$.

(iii) Let $e_1, e_2 \in \mathrm{SE}(\Sigma)$ such that $\mathrm{sym}(e_1) \cap \mathrm{sym}(e_2) = \emptyset$. Then $e = (e_1 \parallel e_2) \in \mathrm{SE}(\Sigma)$.

(iv) Let $e_1 \in \mathrm{SE}(\Sigma)$. Then $e = (e_1)^* \in \mathrm{SE}(\Sigma)$.

For better readability we usually omit the outermost parentheses of an expression; e.g. $(a \rightarrow b)$ is written simply as $a \rightarrow b$. When there is no danger of confusion we leave out also other parentheses, e.g. write $((a \parallel b) \parallel c)$ simply as $a \parallel b \parallel c$.

To each letter (process) $a \in \Sigma$ we associate symbols a_s and a_t denoting the *start* and *termination* of the process a. Let

$$\Sigma_s = \{a_s | a \in \Sigma\} \text{ and } \Sigma_t = \{a_t | a \in \Sigma\}.$$

Then we can interpret the synchronization expressions of $\mathrm{SE}(\Sigma)$ as languages over $\Sigma_s \cup \Sigma_t$.

For $e \in \mathrm{SE}(\Sigma)$ we define $L(e) \subseteq (\Sigma_s \cup \Sigma_t)^*$ inductively according to the structure of the expression e.

(i) For $a \in \Sigma$, $L(a) = a_s a_t$ and $L(\hat{\lambda}) = \lambda$.

(ii) Let $e = e_1 \rightarrow e_2$. Then $L(e) = L(e_1) \cdot L(e_2)$.

(iii) Let $e = e_1 \mid e_2$. Then $L(e) = L(e_1) \cup L(e_2)$.

(iv) Let $e = e_1 \,\&\, e_2$. Then $L(e) = L(e_1) \cap L(e_2)$.

(v) Let $e = e_1 \parallel e_2$. Then $L(e) = \omega_{sh}(L(e_1), L(e_2))$.

(vi) Let $e = (e_1)^*$. Then $L(e) = L(e_1)^*$.

Note that the *join* operation $\parallel$ is defined only for expressions having disjoint alphabets. A language $L \subseteq \Sigma^*$ is said to be a *(Σ-)synchronization language* if there exists $e \in \mathrm{SE}(\Sigma)$ such that $L = L(e)$. The class of synchronization languages is denoted

$$\mathcal{L}(\mathrm{SE}) = \{L(e) \mid e \in \mathrm{SE}(\Sigma),\ \Sigma \text{ is a finite alphabet}\}.$$

Our aim is to characterize the family of Σ-synchronization languages in terms of a subclass of regular languages over the alphabet $\Sigma_s \cup \Sigma_t$.

For $a \in \Sigma$ define the morphism

$$f_a : (\Sigma_s \cup \Sigma_t)^* \longrightarrow \{a_s, a_t\}^* \tag{4.1}$$

by the conditions $f_a(x) = x$ if $x \in \{a_s, a_t\}$ and $f_a(x) = \lambda$ if $x \in (\Sigma_s \cup \Sigma_t) - \{a_s, a_t\}$. A language $L \subseteq \{\Sigma_s \cup \Sigma_t\}^*$ is said to satisfy the *start-termination* condition (or st-condition for short) if

$$\forall a \in \Sigma\ \forall w \in L\ f_a(w) \in (a_s a_t)^*.$$

Languages satisfying the st-condition will also be called st-languages. It is immediate that all synchronization languages satisfy the st-condition. However, clearly there

are even finite st-languages that are not synchronization languages. For instance the singleton language $\{a_s b_s a_t b_t\}$ is not in $\mathcal{L}(SE)$ because a synchronization language containing the word $a_s b_s a_t b_t$ necessarily contains all words in the shuffle $\omega_{sh}(a_s a_t, b_s b_t)$. Next we will consider certain general properties of synchronization languages.

Example 4.3.1 *A natural question about the above definition of SEs is whether the intersection operator & is necessary. This turns out to be the case. Let* $\Sigma = \{a, b, c, d\}$ *and define*

$$e = ((a \rightarrow b) \parallel (c \rightarrow d)) \,\&\, ((a \rightarrow d) \parallel b \parallel c).$$

It turns out that $L(e)$ *cannot be represented in the form* $L(e')$*, where* e' *is an SE that does not contain the operator &. Thus the family of synchronization languages defined without the operator & would not be closed under intersection.*

For the sake of contradiction, assume that $L(e) = L(e')$*, where* e' *does not contain the intersection operation. Since* $L(e)$ *is finite, we can assume that* e' *does not contain the repetition operator* **. Using the distributivity of union, we can write*

$$L(e') = L(e_1) \cup \ldots \cup L(e_m),$$

where e_i *contains only sequencing and join operations* $\rightarrow$ *and* $\parallel$*. Consider*

$$w = a_s c_s a_t b_s c_t d_s b_t d_t.$$

Since $w \in L(e)$ *it follows that* $w \in L(e_i)$ *for some* $i \in \{1, \ldots, m\}$*. We have two possibilities:*

(i) $e_i = \eta_1 \rightarrow \eta_2$, *or,*

(ii) $e_i = \eta_1 \parallel \eta_2$,

where η_1 *and* η_2 *are expressions containing only operations* $\rightarrow$ *and* $\parallel$*.*

(i) Since the word w does not have a decomposition of the form $w_1 w_2$ *where* w_i *satisfies the st-condition,* $i = 1, 2$*, it follows that necessarily* $\eta_1 = \hat{\lambda}$ *or* $\eta_2 = \hat{\lambda}$*. We use then the same argument for* η_2 *or* η_1*, respectively.*

(ii) Since $w \in L(\eta_1 \| \eta_2)$*, it follows that*

$$sym(\eta_1) \cup sym(\eta_2) = \{a, b, c, d\}, \quad sym(\eta_1) \cap sym(\eta_2) = \emptyset. \tag{4.2}$$

The symbols a and b, a and d, c and d are pairwise in sequential relation in words of $L(e)$*. Hence all the above three pairs necessarily belong to* $sym(\eta_1)$ *or to* $sym(\eta_2)$*. This implies that a partition as in (4.2) is possible only if* $\eta_1 = \hat{\lambda}$ *or* $\eta_2 = \hat{\lambda}$*.*

By now it is clear that the family of synchronization languages is a proper subset of the families of regular start-termination languages. Let Σ be an alphabet. Then it is also easy to show that the family of synchronization languages over $\Sigma_s \cup \Sigma_t$ is the smallest language family that contains $\{a_s a_t\}$, for each $a \in \Sigma$, and is closed under concatenation, union, intersection, Kleene*-operation, and shuffle operation where the operands are over disjoint sub-alphabets.

We say that two SEs e_1 and e_2 are equivalent if $L(e_1) = L(e_2)$. Similarly, we say that e_1 includes e_2 if $L(e_2) \subseteq L(e_1)$. From the above observation, we have the following.

Theorem 4.3.1 *Given two SEs e_1 and e_2, we can effectively decide whether e_1 and e_2 are equivalent and whether e_1 includes e_2.*

Example 4.3.2 *The SE $(p \rightarrow q) \mid (p \parallel q)$ can be simplified to $p \parallel q$ since the latter is equivalent to the former. Similarly, the SE $(c \parallel (a \rightarrow b)) \,\&\, ((c \rightarrow b) \parallel a)$ can be simplified to $(c \parallel a) \rightarrow b$.*

Example 4.3.3 *The two SEs $e_1 = (r \parallel (p \rightarrow q))$ and $e_2 = (((p \parallel r) \rightarrow q) \mid (p \rightarrow (q \parallel r)))$ are not equivalent since the word $p_s r_s p_t q_s r_t q_t$ is in $L(e_1)$ and not in $L(e_2)$.*

4.4 Rewriting Rules on Synchronization Languages

Next we will consider closure properties of synchronization languages under certain types of rewriting systems. Our aim is then to show that these properties are sufficient to characterize the class of synchronization languages.

Let $L = L(e)$ be a Σ-synchronization language and consider $w \in L$. Let $a, b \in \Sigma$, $a \neq b$, and let

$$f_a(w) = a_s^{(1)} a_t^{(1)} \cdots a_s^{(m)} a_t^{(m)}, \; f_b(w) = b_s^{(1)} b_t^{(1)} \cdots b_s^{(n)} b_t^{(n)}, \; m, n \geq 0. \qquad (4.3)$$

Here $a^{(i)}$, $b^{(j)}$, $1 \leq i \leq m$, $1 \leq j \leq n$, denote just the different occurrences of the symbols a and b, respectively, in the word w. We say that occurrences $a^{(i)}$ and $b^{(j)}$, $1 \leq i \leq m$, $1 \leq j \leq n$, are *sequential* if in the word w the symbol $a_t^{(i)}$ precedes $b_s^{(j)}$ or $b_t^{(j)}$ precedes $a_s^{(i)}$. Intuitively, this means that the process $a^{(i)}$ has terminated before the process $b^{(j)}$ starts, or vice versa. If $a^{(i)}$ and $b^{(j)}$ are not sequential, we say that $a^{(i)}$

and $b^{(j)}$ are *parallel*, this is denoted $a^{(i)}\Psi_{par}b^{(j)}$. If $a^{(i)}\Psi_{par}b^{(j)}$ then the occurrences $a^{(i)}$ and $b^{(j)}$ necessarily correspond to different arguments of the join-operator ($\|$) in some subexpression of e. Thus the symbols $a_s^{(i)}$, $a_t^{(i)}$ commute with the symbols $b_s^{(j)}$, $b_t^{(j)}$ in the word w. That is, the word obtained from w by commuting one of the symbols $a_s^{(i)}$, $a_t^{(i)}$ with $b_s^{(j)}$ or $b_t^{(j)}$ belongs to the language $L(e)$. From these observations we obtain immediately the following lemma.

Lemma 4.4.1 *Let Σ be a finite alphabet. Consider the Thue system $R_1(\Sigma) \subseteq (\Sigma_s \cup \Sigma_t)^* \times (\Sigma_s \cup \Sigma_t)^*$ consisting of the following rules, for all $a, b \in \Sigma$, $a \neq b$:*

(i) $a_s b_s \rightarrow b_s a_s$,

(ii) $a_t b_t \rightarrow b_t a_t$,

(iii) $a_s b_t \rightarrow b_t a_s$.

Let L be a Σ-synchronization language. Then L is closed under $R_1(\Sigma)$, i.e., $\Delta_{R_1(\Sigma)}(L) \subseteq L$. □

In the notation of (4.3), assume that $a^{(i)}\Psi_{par}b^{(j)}$, $i \in \{1,\ldots,m\}$, $j \in \{1,\ldots,n\}$, and that we can write

$$w = u_1 a_s^{(i)} u_2 b_s^{(j)} u_3 b_t^{(j)} u_4 a_t^{(i)} u_5.$$

Then it does not in general follow that one could exchange the positions of $a_s^{(i)}$ and $b_s^{(j)}$ (or of $b_t^{(j)}$ and $a_t^{(i)}$.) The occurrence b_j may namely be in a sequential relation with some symbols appearing in the subword u_2 (or u_4). The rewrite rules of $R_1(\Sigma)$ as defined in Lemma 4.4.1 permute only consecutive symbols.

Closure under the Thue system $R_1(\Sigma)$ (together with the st-condition) is not sufficient to characterize the synchronization languages. There are even finite st-languages closed under $R_1(\Sigma)$ that do not belong to $\mathcal{L}(SE)$. This can be shown by the following example.

Example 4.4.1 *Let $\Sigma = \{a, b\}$ and choose $w = a_s b_s a_t a_s b_t b_s a_t b_t$. Define $L = \Delta^*_{R_1(\Sigma)}(w)$. Thus L is an st-language closed under $R_1(\Sigma)$. However, L is not a synchronization language, i.e., L cannot be expressed by any SE. This can be proved by showing that L is not closed under the rewriting rule*

$$a_t a_s b_t b_s \rightarrow b_t b_s a_t a_s,$$

which is stated in the next lemma.

Definition 4.4.1 *Define* $R_2(\Sigma) \subseteq (\Sigma_s \cup \Sigma_t)^* \times (\Sigma_s \cup \Sigma_t)^*$ *as follows. Let* $X = \{x_s, x_t, y_s, y_t\}$ *where* $X \cap (\Sigma_s \cup \Sigma_t) = \emptyset$. *For each sequence* $a_1, \ldots, a_m, b_1, \ldots, b_n$ *of pairwise distinct elements of* Σ, $m, n \geq 1$, *consider the morphism defined by the conditions*

$$h(x_s) = a_{1s} \cdots a_{ms}, \quad h(x_t) = a_{1t} \cdots a_{mt}, \quad h(y_s) = b_{1s} \cdots b_{ns}, \quad h(y_t) = b_{1t} \cdots b_{nt}.$$

Then $R_2(\Sigma)$ *has the rule:*

$$h(x_t x_s y_t y_s) \to h(y_t y_s x_t x_s). \tag{4.4}$$

Here a_{iz}, b_{jz} *stand for* $(a_i)_z$, $(b_j)_z$, $1 \leq i \leq m$, $1 \leq j \leq n$, $z \in \{s, t\}$.

Lemma 4.4.2 *Let* L *be a* Σ*-synchronization language and let* $R_2(\Sigma)$ *be defined as above. Then* L *is closed under* $R_2(\Sigma)$.

Proof. We say that the *rank* of the rewrite rule (4.4) is $m + n$. For $M \geq 1$, let $r(M)$ denote the subset of $R_2(\Sigma)$ consisting of all rules of rank at most M. Note that $r(\#\Sigma) = R_2(\Sigma)$. Using induction on M we prove that

$$\Delta_{r(M)}(L(e)) \subseteq L(e), \; M = 2, \ldots, \#\Sigma, \tag{4.5}$$

for an arbitrary language $L(e)$, $e \in \mathrm{SE}(\Sigma)$.

(i) First consider the case $M = 2$. We prove that (4.5) holds using induction on the structure of the synchronization expression e. Clearly (4.5) is valid for $e \in \Sigma \cup \{\hat{\lambda}\}$. Assume that (4.5) (with $M = 2$) holds for $e_1, e_2 \in \mathrm{SE}(\Sigma)$. Thus $L(e_i)$ is closed under $r(2)$, $i = 1, 2$, which implies that (4.5) holds also for $e_1|e_2$ and $e_1 \& e_2$. Consider the case $e = e_1 \to e_2$. Let $w \in L(e)$ have a subword that is a left side of a rule of $r(2)$, i.e.,

$$w = u a_t a_s b_t b_s v, \; a, b \in \Sigma, \; a \neq b. \tag{4.6}$$

On the other hand, $w = w_1 w_2$ where $w_i \in L(e_i)$, $i = 1, 2$. Since $L(e_i)$ satisfies the st-condition, $i = 1, 2$, $a_t a_s b_t b_s$ is a subword of w_1 or w_2. By the induction assumption, $u b_t b_s a_t a_s v \in L(e_1) \cdot L(e_2)$. The case where $e = (e_1)^*$ is similar.

The remaining possibility is that $e = e_1 || e_2$ and again consider $w \in L(e)$ as in (4.6). Thus $w = \omega_{sh}(w_1, w_2)$, $w_i \in L(e_i)$, $i = 1, 2$. If $a_t a_s b_t b_s$ is a subword of w_i, $i \in \{1, 2\}$, we are done by the inductive assumption. Since $\mathrm{sym}(e_1) \cap \mathrm{sym}(e_2) = \emptyset$, the only other possibility is that $a_t a_s \in \mathrm{sub}(w_1)$, $b_t b_s \in \mathrm{sub}(w_2)$ or vice versa. Immediately from the definition of the shuffle operation it follows that $u b_t b_s a_t a_s v \in \omega_{sh}(w_1, w_2) \subseteq L(e)$.

(ii) Assume then that (4.5) holds for $M = k-1 < \#\Sigma$ and all $e \in \mathrm{SE}(\Sigma)$. We prove that $L(e)$ is closed under $r(k)$ for all $e \in \mathrm{SE}(\Sigma)$ using induction on the structure of the expression e. As in (i) above, we see that the only nontrivial case is the join-operation. Let $e = e_1||e_2$ where $L(e_i)$ is closed under $r(k)$, $i = 1,2$. Let $w \in L(e)$ have an occurrence of a left side of a rewrite rule of rank at most k. Thus we can write

$$w = ua_t^{(1)} \cdots a_t^{(m)} a_s^{(1)} \cdots a_s^{(m)} b_t^{(1)} \cdots b_t^{(t)} b_s^{(1)} \cdots b_s^{(n)} v, \quad m+n \leq k.$$

Now $w = \omega_{sh}(w_1, w_2)$, where $w_i \in L(e_i)$, $i = 1,2$. Since $\mathrm{sym}(e_1) \cap \mathrm{sym}(e_2) = \emptyset$, both symbols $a_s^{(i)}$, $a_t^{(i)}$, $i \in \{1,\ldots,m\}$ (respectively, $b_s^{(j)}$, $b_t^{(j)}$, $j \in \{1,\ldots,n\}$) belong to either w_1 or w_2. Thus there exist disjoint partitions

$$\{1,\ldots,m\} = \{A_1,\ldots,A_\alpha\} \cup \{B_1,\ldots,B_\beta\},$$

$$\{1,\ldots,n\} = \{C_1,\ldots,C_\gamma\} \cup \{D_1,\ldots,D_\delta\},$$

such that

$$a_t^{(A_1)} \cdots a_t^{(A_\alpha)} a_s^{(A_1)} \cdots a_s^{(A_\alpha)} b_t^{(C_1)} \cdots b_t^{(C_\gamma)} b_s^{(C_1)} \cdots b_s^{(C_\gamma)} \in \mathrm{sub}(w_1),$$

and

$$a_t^{(B_1)} \cdots a_t^{(B_\beta)} a_s^{(B_1)} \cdots a_s^{(B_\beta)} b_t^{(D_1)} \cdots b_t^{(D_\delta)} b_s^{(D_1)} \cdots b_s^{(D_\delta)} \in \mathrm{sub}(w_2).$$

We can assume that

$$\alpha + \gamma \geq 1 \text{ and } \beta + \delta \geq 1. \tag{4.7}$$

Otherwise, $a_t^{(1)} \cdots a_t^{(m)} a_s^{(1)} \cdots a_s^{(m)} b_t^{(1)} \cdots b_t^{(n)} b_s^{(1)} \cdots b_s^{(n)}$ is a subword of w_1 or of w_2, and we are done since $L(e_i)$, $i = 1,2$, is closed under $r(k)$. Furthermore, we can assume that

$$\alpha + \delta \geq 1. \tag{4.8}$$

If $\alpha + \delta = 0$, then $a_t^{(1)} \cdots a_t^{(m)} a_s^{(1)} \cdots a_s^{(m)}$ is a subword of w_2 and $b_t^{(1)} \cdots b_t^{(n)} b_s^{(1)} \cdots b_s^{(n)}$ is a subword of w_1 and it follows again immediately that

$$ub_t^{(1)} \cdots b_t^{(n)} b_s^{(1)} \cdots b_s^{(n)} a_t^{(1)} \cdots a_t^{(m)} a_s^{(1)} \cdots a_s^{(m)} v \in \omega_{sh}(w_1, w_2) \subseteq L(e).$$

Now by the definition of the shuffle operation we have:

$$ua_t^{(B_1)} \cdots a_t^{(B_\beta)} a_s^{(B_1)} \cdots a_s^{(B_\beta)} b_t^{(D_1)} \cdots b_t^{(D_\delta)} b_s^{(D_1)} \cdots b_s^{(D_\delta)} a_t^{(A_1)} \cdots a_t^{(A_\alpha)} a_s^{(A_1)} \cdots a_s^{(A_\alpha)} b_t^{(C_1)} \cdots$$
$$b_t^{(C_\gamma)} b_s^{(C_1)} \cdots b_s^{(C_\gamma)} v \in \omega_{sh}(w_1, w_2) \subseteq L(e).$$

By (4.7), $\beta + \delta < k$ and $\alpha + \gamma < k$, and thus $L(e)$ is closed under the rules

$$a_t^{(B_1)} \dots a_t^{(B_\beta)} a_s^{(B_1)} \dots a_s^{(B_\beta)} b_t^{(D_1)} \dots b_t^{(D_\delta)} b_s^{(D_1)} \dots b_s^{(D_\delta)} \rightarrow$$
$$b_t^{(D_1)} \dots b_t^{(D_\delta)} b_s^{(D_1)} \dots b_s^{(D_\delta)} a_t^{(B_1)} \dots a_t^{(B_\beta)} a_s^{(B_1)} \dots a_s^{(B_\beta)},$$

and,

$$a_t^{(A_1)} \dots a_t^{(A_\alpha)} a_s^{(A_1)} \dots a_s^{(A_\alpha)} b_t^{(C_1)} \dots b_t^{(C_\gamma)} b_s^{(C_1)} \dots b_s^{(C_\gamma)} \rightarrow$$
$$b_t^{(C_1)} \dots b_t^{(C_\gamma)} b_s^{(C_1)} \dots b_s^{(C_\gamma)} a_t^{(A_1)} \dots a_t^{(A_\alpha)} a_s^{(A_1)} \dots a_s^{(A_\alpha)}.$$

Thus,

$$ub_t^{(D_1)} \dots b_t^{(D_\delta)} b_s^{(D_1)} \dots b_s^{(D_\delta)} a_t^{(B_1)} \dots a_t^{(B_\beta)} a_s^{(B_1)} \dots a_s^{(B_\beta)} b_t^{(C_1)} \dots b_t^{(C_\gamma)} b_s^{(C_1)} \dots b_s^{(C_\gamma)} a_t^{(A_1)} \dots$$
$$a_t^{(A_\alpha)} a_s^{(A_1)} \dots a_s^{(A_\alpha)} v \in L(e).$$

By (4.8), $\beta + \gamma < k$, and again by the inductive assumption

$$ub_t^{(D_1)} \dots b_t^{(D_\delta)} b_s^{(D_1)} \dots b_s^{(D_\delta)} b_t^{(C_1)} \dots b_t^{(C_\gamma)} b_s^{(C_1)} \dots b_s^{(C_\gamma)} a_t^{(B_1)} \dots a_t^{(B_\beta)} a_s^{(B_1)} \dots$$
$$a_s^{(B_\beta)} a_t^{(A_1)} \dots a_t^{(A_\alpha)} a_s^{(A_1)} \dots a_s^{(A_\alpha)} v \in L(e).$$

By Lemma 4.4.1, $L(e)$ is closed under $R_1(\Sigma)$. Thus we have,

$$ub_t^{(1)} \dots b_t^{(n)} b_s^{(1)} \dots b_s^{(n)} a_t^{(1)} \dots a_t^{(m)} a_s^{(1)} \dots a_s^{(m)} v \in L(e).$$

This concludes the proof. □

For a given alphabet Σ let

$$R(\Sigma) = R_1(\Sigma) \cup R_2(\Sigma),$$

where $R_1(\Sigma)$ and $R_2(\Sigma)$ are as in Lemmas 4.4.1 and 4.4.2. Then combining the above lemmas we have:

Theorem 4.4.1 *An arbitrary Σ-synchronization language is closed under $R(\Sigma)$.*

The following example shows that there exist regular st-languages over an alphabet $\Sigma_s \cup \Sigma_t$ such that their closure under $R(\Sigma)$ is not even regular.

Example 4.4.2 *Let $\Sigma = \{a, b\}$. Let L be the language defined by the regular expression*

$$a_s(b_s a_t a_s b_t)^* a_t. \tag{4.9}$$

It is easy to see that for all $m \geq 0$, $a_s(b_s a_t a_s b_t)^m a_t \rightarrow^*_{R(\Sigma)} (a_s a_t)^{m+1}(b_s b_t)^m$. Thus

$$\Delta^*_{R(\Sigma}(L) \cap (a_s a_t)^*(b_s b_t)^* = \{(a_s a_t)^{m+1}(b_s b_t)^m \mid m \geq 0\}.$$

It follows that $\Delta^*_{R(\Sigma)}(L) \notin REG(\Sigma)$.

We want to characterize the synchronization languages in terms of a subclass of regular languages (expressions) that is closed under $R(\Sigma)$. In view of the above example it is natural to prohibit cycles of the type $b_s a_t a_s b_t$, where the start- and termination-symbols corresponding to a process a occur in consecutive repetitions of the cycle. For this purpose we associate, with a given regular expression $\alpha \in \mathrm{reg}(\Sigma)$, a type $\tau(\alpha)$. Using the notion of type we first define consistency conditions for regular expressions that guarantee that the corresponding languages satisfy the st-condition. Well-formed regular expressions are defined by restricting the type of the argument of the *-operation. Well-formed regular expressions do not allow cycles like the one in (4.9).

First we need some preliminaries. Let

$$A = \{s,t\}^2 \cup \{0,1\} \tag{4.10}$$

and define an associative binary operation $\otimes \subseteq A \times A$ as follows. The elements 0 and 1 are, respectively, the zero and identity of the semigroup $(A, \otimes)$, i.e., $0 \otimes x = x \otimes 0 = 0$ and $1 \otimes x = x \otimes 1 = x$ for all $x \in A$. Let $(x_1, x_2), (y_1, y_2) \in \{s,t\}^2$. Then

$$(x_1, x_2) \otimes (y_1, y_2) = \begin{cases} (x_1, y_2), & \text{if } x_2 \neq y_1; \\ 0, & \text{if } x_2 = y_1. \end{cases}$$

The operation $\otimes$ is extended for subsets of A by setting

$$X \otimes Y = \{x \otimes y \mid x \in X,\ y \in Y\} X, Y \in \wp(A),$$

and furthermore, componentwise for cartesian products of $\wp(A)$:

$$(X_1, \ldots, X_n) \otimes (Y_1, \ldots, Y_n) = (X_1 \otimes Y_1, \ldots, X_n \otimes Y_n),$$

$n \geq 1$, $X_i, Y_i \in \wp(A)$, $i = 1, \ldots, n$.

Definition 4.4.2 *Let $\Sigma = \{a_1, \ldots, a_k\}$, $k \geq 1$, and let A be as in (4.10). The* type *of a regular expression $\alpha \in reg\ (\Sigma_s \cup \Sigma_t)$, $\tau(\alpha)$, is a k-tuple of elements of $\wp(A)$ defined as follows.*

(i) Let $\alpha = (a_i)_z$, $1 \leq i \leq k$, $z \in \{s,t\}$. Then $\tau(\alpha)_i = (z,z)$ and $\tau(\alpha)_j = 1$ for $j \neq i$.

(ii) Assume that $\alpha = \alpha_1 \cdot \alpha_2$. Then $\tau(\alpha) = \tau(\alpha_1) \otimes \tau(\alpha_2)$.

(iii) *Let* $\alpha = \alpha_1 + \alpha_2$, $\tau(\alpha_1) = (X_1, \ldots, X_k)$, $\tau(\alpha_2) = (Y_1, \ldots, Y_k)$. *Then*

$$\tau(\alpha) = (X_1 \cup Y_1, \ldots, X_k \cup Y_k).$$

(iv) *For* $\alpha = (\alpha_1)^*$, *we define* $\tau(\alpha)$ *by setting* $[\tau(\alpha)]_i = [\tau(\alpha_1) \otimes \tau(\alpha_1)]_i \cup \{1\}$, $i = 1, \ldots, k$.

A regular expression $\alpha \in reg(\Sigma_s \cup \Sigma_t)$ *is said to be* consistent *iff* $[\tau(\alpha)]_i \subseteq \{1, (s,t)\}$ *for all* $i = 1, \ldots, k$.

The meaning of a type $\tau(\alpha) = (X_1, \ldots, X_k)$ can be explained intuitively as follows. If $0 \in X_i$, then there exists a word $w \in L(\alpha)$ such that $f_{a_i}(w)$ contains a prohibited subword $(a_i)_s(a_i)_s$ or $(a_i)_t(a_i)_t$. (Here f_{a_i} is as in (4.1).) In this case, α cannot be a subexpression of any consistent regular expression. If $0 \notin X_i$, then X_i consists of all pairs (x,y), $x, y \in \{s,t\}$, such that there exists $w \in L(\alpha)$ with $\text{first}[f_{a_i}(w)] = (a_i)_x$ and $\text{last}[f_{a_i}(w)] = (a_i)_y$. These observations are formalized in the lemma below.

Lemma 4.4.3 *Let* $\Sigma = \{a_1, \ldots, a_k\}$, $k \geq 1$. *Let* $\alpha \in reg(\Sigma_s \cup \Sigma_t)$ *and* $i \in \{1, \ldots, k\}$. *Denote* $[\tau(\alpha)]_i = X (\subseteq A)$. *To simplify the notation, we denote* a_i *just by* a. *Define* $C_1 = (a_s a_t)^+$, $C_2 = (a_s a_t)^* a_s$, $C_3 = a_t(a_s a_t)^*$, $C_4 = a_t(a_s a_t)^* a_s$, *and,* $C_5 = \{a_s, a_t\}^* \cdot \{a_s a_s, a_t a_t\} \cdot \{a_s, a_t\}^*$.

Let f_a *be as in (4.1) and let* $L = f_a(L(\alpha))$. *Then the following holds:*

(i) $1 \in X$ *iff* $\lambda \in L$.

(ii) $(s,t) \in X$ *iff* $C_1 \cap L \neq \emptyset$.

(iii) $(s,s) \in X$ *iff* $C_2 \cap L \neq \emptyset$.

(iv) $(t,t) \in X$ *iff* $C_3 \cap L \neq \emptyset$.

(v) $(t,s) \in X$ *iff* $C_4 \cap L \neq \emptyset$.

(vi) $0 \in X$ *iff* $C_5 \cap L \neq \emptyset$.

Theorem 4.4.2 *Let* $\alpha \in reg(\Sigma_s \cup \Sigma_t)$. *Then* $L(\alpha)$ *satisfies the st-condition iff* α *is consistent.*

Proof. Denote $\Sigma = \{a_1, \ldots, a_k\}$, $k \geq 1$. Assume that $L(\alpha)$ is an st-language. Then for every $i \in \{1, \ldots, k\}$ $f_{a_i}(L(\alpha)) \subseteq ((a_i)_s(a_i)_t)^*$. Thus by Lemma 4.4.3, $[\tau(\alpha)]_i \subseteq \{1, (s,t)\}$, $i = 1, \ldots, k$; i.e., α is consistent. The converse implication follows similarly from Lemma 4.4.3. □

Clearly all finite synchronization languages can be represented by consistent regular expressions. Conversely, closure under the rewriting system $R(\Sigma)$ is a necessary condition (by Theorem 4.4.1) for a language to be a synchronization language. Given a consistent word $w \in (\Sigma_s \cup \Sigma_t)^*$ we *try to* construct a synchronization expression e such that $L(e) = \Delta^*_{R(\Sigma)}(w)$, which implies then that all finite $R(\Sigma)$-closed st-languages are synchronization languages.

A consistent word $w \in (\Sigma_s \cup \Sigma_t)^*$ is said to be *reducible* if w has a consistent proper prefix. Otherwise, w is said to be irreducible. For instance, the word w considered in Example 4.3.1 is irreducible. An arbitrary consistent word w has a unique decomposition

$$w = w_1 \cdots w_m,$$

where each w_i is irreducible.

First we consider the case where the cardinality of Σ is 2.

Lemma 4.4.4 *Let $\Sigma = \{a, b\}$. Let $w \in (\Sigma_s \cup \Sigma_t)^*$ be consistent and irreducible and assume that $|w| \geq 4$. We claim that $w \in L$, where*

$$L = \{a_s b_s, b_s a_s\}\{b_t b_s, a_t a_s\}^*\{a_t b_t, b_t a_t\}. \tag{4.11}$$

Proof. Clearly the prefix of w of length 2 is either $a_s b_s$ or $b_s a_s$. Let w' be a prefix of w such that $1 \leq |w'| < |w| - 1$. We observe that the $(|w'| + 1)$st symbol x of w always has to be chosen as follows:

(I) If $\tau(w') = [(s,s),(s,s)]$, then (i) $x = a_t$ and $\tau(w'x) = [(s,t),(s,s)]$, or (ii) $x = b_t$ and $\tau(w'x) = [(s,s),(s,t)]$. (A choice $x \in \{a_s, b_s\}$ violates the consistency condition.)

(II) (i) If $\tau(w') = [(s,t),(s,s)]$, then $x = a_s$ and $\tau(w'x) = [(s,s),(s,s)]$. (If $x = b_t$, then $w'x$ is a consistent proper prefix of w.)
(ii) If $\tau(w'x) = [(s,s),(s,t)]$, then $x = b_s$ and $\tau(w'x) = [(s,s),(s,s)]$.

Starting from $a_s b_s$ or $b_s a_s$ the rules (I) and (II) generate exactly all the words of L. The process is ended by choosing $x = b_t$ or $x = a_t$ in (II) (i) and (ii), respectively. □

We introduce the following notation. Let $a \in \Sigma$ and let $e \in \mathrm{SE}(\Sigma)$ be obtained by sequencing $m \geq 0$ copies of a, i.e., $e = a \rightarrow \cdots \rightarrow a$. Then we denote

$$e = a^{(m,\rightarrow)}.$$

The expression $a^{(0,\rightarrow)}$ is interpreted to stand for $\hat{\lambda}$.

Lemma 4.4.5 *Let $\Sigma = \{a, b\}$. For a consistent word w in $(\Sigma_s \cup \Sigma_t)^*$, and $c \in \{a, b\}$, we denote by w_c the number of symbols c_s (or c_t) appearing in the word w. Let $w \in (\Sigma_s \cup \Sigma_t)^*$ be (consistent and) irreducible. We claim that*

$$\Delta^*_{R(\Sigma)}(w) = a^{(w_a,\rightarrow)} \parallel b^{(w_b,\rightarrow)}. \tag{4.12}$$

Proof. Clearly the claim holds if $w \in \{a_s a_t, b_s b_t\}$. In the following we assume that $|w| \geq 4$. By Lemma 4.4.4, $w \in L$, where L is as in (4.11). Denote

$$L_m = \{w \in L \mid |w| \leq 2m\}, \quad m \geq 0.$$

We prove that (4.12) holds for all $w \in L_m$ using induction on m. If $w \in L_2$, then clearly $\Delta^*_{R(\Sigma)}(w) = \omega_{sh}(a_s a_t, b_s b_t)$. Assume that (4.12) holds for all $w \in L_m$, $m \geq 1$, and consider $w' \in L_{m+1} - L_m$; i.e., $|w'| = 2m + 2$. The word w' is obtained from some $w \in L_m$ by replacing the suffix $b_t a_t$ with the word (i) $b_t b_s b_t a_t$ or (ii) $a_t a_s b_t a_t$. (Of course the last symbols b_t and a_t can be permuted.) We consider (i), the case of (ii) is quite similar. We can write

$$w' = (w_1 b_t) b_s b_t a_t, \; w = (w_1 b_t) a_t,$$

for some $w_1 \in (\Sigma_s \cup \Sigma_t)^*$. Let $u \in a^{(w'_a,\rightarrow)} \parallel b^{(w'_b,\rightarrow)}$ be arbitrary. The word u is obtained from a word $v \in a^{(w_a,\rightarrow)} \parallel b^{(w_b,\rightarrow)}$ by adding the symbols b_s and b_t (in this order) in arbitrary positions after the last occurrence of b_t in the word v. By the inductive assumption, the rules of $R(\Sigma)$ produce v from w. Intuitively, one can think that this is done by moving the b-symbols with respect to the a-symbols. Now the word u can be obtained from w' by simulating the reduction of w to v and if necessary moving the new (last) b_s-symbol of w' to the left together with the last b_t-symbol of w using rules $a_t a_s b_t b_s \rightarrow b_t b_s a_t a_s$. Note that otherwise the new symbol b_s could not "cross over" a_t-symbols. Thus (4.12) holds also for w'. □

Lemma 4.4.6 *Let $\Sigma = \{a, b\}$ and let $w \in (\Sigma_s \cup \Sigma_t)^*$ be consistent. Then there exist nonnegative integers $m_1, \ldots, m_k$, $n_1, \ldots, n_k$, $k \geq 0$ such that*

$$\Delta^*_{R(\Sigma)}(w) = L(e[w]),$$

where $e[w] \in SE(\Sigma)$ is of the form

$$e[w] = (a^{(m_1,\rightarrow)} \parallel b^{(n_1,\rightarrow)}) \rightarrow \cdots \rightarrow (a^{(m_k,\rightarrow)} \parallel b^{(n_k,\rightarrow)}). \tag{4.13}$$

Proof. The word w has a (unique) decomposition $w_1 \cdots w_k$, $k \geq 0$, where w_i is irreducible, $i = 1, \ldots, k$. By Lemma 4.4.5, $\Delta^*_{R(\Sigma)}(w_i) = L(e_i)$, where $e_i = a^{(m_i,\rightarrow)} || b^{(n_i,\rightarrow)}$, $m_i, n_i \geq 0$, $i = 1, \ldots, k$. Since each w_i is consistent, we have

$$\Delta^*_{R(\Sigma)}(w) = \Delta^*_{R(\Sigma)}(w_1) \cdots \Delta^*_{R(\Sigma)}(w_k) = L(e_1) \cdots L(e_k) = L(e_1 \rightarrow \cdots \rightarrow e_k).$$

□

Let Σ be a finite alphabet and $a, b \in \Sigma$, $a \neq b$. The morphism $g_{a,b} : (\Sigma_s \cup \Sigma_t)^* \longrightarrow \{a_s, a_t, b_s, b_t\}^*$ is defined by setting $g_{a,b}(x) = x$ if $x \in \{a_s, a_t, b_s, b_t\}$ and $g_{a,b}(x) = \lambda$ if $x \notin \{a_s, a_t, b_s, b_t\}$. For $w \in \Sigma^*$, $g_{a,b}(w)$ is just the projection of the word w to the alphabet $\{a_s, a_t, b_s, b_t\}$.

If $w \in \Sigma^*$ is consistent, then clearly also $g_{a,b}(w)$ is consistent for all $a, b \in \Sigma^*$. For a consistent $w \in \Sigma^*$ and $a, b \in \Sigma$, $a \neq b$, let $e[g_{a,b}(w)] \in \mathrm{SE}(\{a, b\})$ be defined as in (4.13). Let $c \in \Sigma$. For a consistent word $w \in (\Sigma_s \cup \Sigma_t)^*$ we denote by w_c the number of occurrences of the symbol c_s (or c_t) in the word w.

Claim 4.4.1 *Let $\#\Sigma \geq 2$ and $w \in \Sigma^*$ be consistent. Assume that $a, b \in \Sigma$, $a \neq b$ are fixed. Denote $\Sigma - \{a, b\} = \{c_1, \ldots, c_k\}$, $k \geq 0$. Define*

$$e[w, a, b] = e[g_{a,b}(w)] \,||\, c_1^{(w_{c_1},\rightarrow)} \,||\, \cdots \,||\, c_k^{(w_{c_k},\rightarrow)}.$$

We claim that

$$\Delta^*_{R(\Sigma)}(w) = \bigcap_{a,b \in \Sigma, a \neq b} e[w, a, b]. \tag{4.14}$$

If $\#\Sigma = 2$, this is just the result of Lemma 4.4.6. Clearly, the inclusion from left to right holds in (4.14).

Definition 4.4.3 *A consistent regular expression α is said to be* well-formed *if the following condition holds: If $(\beta)^*$ is a subexpression of α, then β is consistent.*

In constructing a well-formed expression one is allowed to use the *-operation only with a consistent argument. The motivation behind this restriction is to prohibit anomalies as in (4.9). Note that the expression (4.9) is consistent but the $R(\Sigma)$-closure of the corresponding language is not even regular. Using a similar example one sees that the $R(\Sigma)$-closure of a language defined by an arbitrary consistent regular expression need not be even context-free.

For a finite alphabet Σ, denote by $\mathcal{L}_{wc}(\Sigma)$ the family of languages over Σ defined by well-formed consistent regular expressions. The following result holds:

Theorem 4.4.3 *If* $L \in \mathcal{L}_{wc}(\Sigma)$ *then* $\Delta^*_{R_1(\Sigma)}(L) \in \mathcal{L}_{wc}(\Sigma)$. *Here* $R_1(\Sigma)$ *is defined as in Lemma 4.4.1.*

We conjecture that for every $L \in \mathcal{L}_{wc}(\Sigma)$, the $R(\Sigma)$-closure of L, i.e., $\Delta^*_{R(\Sigma)}(L)$, is also in $\mathcal{L}_{wc}(\Sigma)$. We also conjecture that the family of Σ-synchronization languages consists of exactly all languages $\Delta^*_{R(\Sigma)}(L)$ where $L \in \mathcal{L}_{wc}(\Sigma)$; that is, the Σ-synchronization languages can be characterized as the $R(\Sigma)$-closures of languages defined by well-formed consistent regular expressions. The result appears to hold at least for finite languages but we have not yet worked out a complete proof.

Chapter 5

Implementation of SEs

In this chapter, we describe an implementation of synchronization expressions. To avoid unnecessary details, we focus only on the essential feature: using reversed alternating finite automata to implement synchronization expressions. From the previous chapter, it is clear that an SE can be expressed by a finite automaton. However, such an automaton may have a large number of states. For example, the very simple SE

$$a \parallel b$$

corresponds to the following st-language:

$$a_s b_s a_t b_t + a_s b_s b_t a_t + a_s a_t b_s b_t + b_s a_s b_t a_t + b_s a_s a_t b_t + b_s b_t a_s a_t.$$

and a DFA accepting the language contains at least nine states. To reduce the complexity in the number of states, reversed alternating finite automata are used as the essential part of our implementation of synchronization expressions.

In the following discussion, we first look at the basic concept of alternating finite automata and define our reversed alternating finite automata. Then we describe, relatively formally, the transformation of synchronization expressions into reversed alternating finite automata.

5.1 Alternating Finite Automata

Let us first review the basic concept of alternating finite automata [29, 40]. Alternating automata are a generalization of non-deterministic automata in the following sense: if in a given state q, the automaton reads an input symbol a, it will activate all

states of the automaton to work on the remaining part of the input in parallel. Once the states have completed their tasks, q will evaluate their results, using a Boolean function and pass on the the resulting value to the state by which it was activated. A word w is accepted if the starting state computes the value of 1. Otherwise, it is rejected.

Definition 5.1.1 *An alternating finite automaton (AFA) is a quintuple* (Q, Σ, s, F, g) *with the following properties:*

a) Q *is a finite set, the set of states;*

b) Σ *is an alphabet, the input alphabet;*

c) $s \in Q$ *is the starting state;*

d) $F \subseteq Q$ *is the set of final states;*

e) g *is a mapping of* Q *into the set of all mappings of* $\Sigma \times B^Q$ *into* B.

Define the sequential behavior of an AFA. For $q \in Q$, let

$$g_q : \Sigma \times B^Q \to B : (a, u) \mapsto g(q)(a, u)$$

where $a \in \Sigma$ and $u \in B^Q$.

The mapping can take different forms; for example,

$$g(a)(q)(u) = g_q(a)(u) = g_q(a, u)$$

for $a \in \Sigma$, $q \in Q$, and $u \in B^Q$.

Let $f \in B^Q$ be defined by the condition

$$f_q = 1 \leftrightarrow q \in F.$$

Then f is called the characteristic vector of F.

We extend g to a mapping $\Sigma \times B^Q \to B^Q$ as follows:

$$g(w, u) = \begin{cases} u & \text{if } w = \lambda, \\ g(a, g(x, u)) & \text{if } w = ax, a \in \Sigma \text{ and } x \in \Sigma^*. \end{cases}$$

Then for each $q \in Q$,

$$g_q(w, u) = \begin{cases} u_q & \text{if } w = \epsilon, \\ g_q(a, g(x, u)) & \text{if } w = ax, a \in \Sigma \text{ and } x \in \Sigma^*. \end{cases}$$

Definition 5.1.2 *Given an AFA $A_a = (Q, \Sigma, s, F, g)$, A word $w \in \Sigma$ is accepted by A_a if and only if $g_s(w, f) = 1$. The language accepted by A_a is the set $L(A_a) = \{w \mid w \in \Sigma^* \wedge g_s(w, f) = 1\}$.*

Now we show how a DFA or NFA can be represented by an AFA. Let $A = (Q, \Sigma, \delta, s, F)$ be an NFA. Then an AFA $A_a = (Q, \Sigma, s, F, g)$ is defined as follows:

$$g_q(a, u) = 0 \leftrightarrow u_p = 0 \; \forall p, \; p \in Q \text{ with } (q, a, p) \in \delta$$

where $q \in Q, a \in \Sigma$ and $u \in B^Q$.

An AFA is an s-AFA if the initial state s cannot be reached in any computation. This is formally defined as: for every $a \in \Sigma$ and every $u \in B^Q$, $g(a, u)$ does not depend on u_s. It is clear that for any AFA of k states, one can construct an equivalent s-AFA with at most $k + 1$ states.

We have the following result from [40]:

Theorem 5.1.1 *L is accepted by an s-AFA with $k+1$ states if and only if L^R is accepted by a DFA with 2^k states.*

In [40], the operations of union, intersection, and negation of AFA have been defined. However, AFA are not directly suitable for our purpose. Note that the acceptance of a word by an AFA is actually computed reversely. For a synchronization expression, we need to implement a control that accepts any valid prefix of a word in the corresponding synchronization language. So, instead of using AFA, we use reversed AFA to implement synchronization languages.

5.2 Reversed Alternating Finite Automata

First, let us give the definition of a reversed s-AFA:

Definition 5.2.1 *A reversed s-AFA (rs-AFA) is a quintuple (Q, Σ, S, q_f, g) where Q, Σ, and g are defined as in an s-AFA; $S \subseteq Q$ is the set of starting states; and $q_f \in Q$ is the final state.*

Let $B = \{0, 1\}$ and $I \in B^Q$ be the initialization vector, such that $I_q = 1$ iff $q \in S$. We extend g to a mapping $\Sigma^* \times B^Q \rightarrow B^Q$ as follows:

$$g(w, u) = \begin{cases} u & \text{if } w = \varepsilon, \\ g(a, g(x, u)) & \text{if } w = xa \text{ with } x \in \Sigma^* \text{ and } a \in \Sigma. \end{cases}$$

Similarly, for each $q \in Q$,

$$g_q(w,u) = \begin{cases} u_q & \text{if } w = \lambda, \\ g_q(a, g(x,u)) & \text{if } w = xa \text{ with } x \in \Sigma^* \text{ and } a \in \Sigma. \end{cases}$$

I.e., $g_q(w,u) = (g(w,u))_q$.

A word $w \in \Sigma$ is said to be accepted by A if and only if $g_{q_f}(w,I) = 1$. The language accepted by A is the set $L(A) = \{w \mid w \in \Sigma^* \wedge g_{q_f}(w,I) = 1\}$.

The following theorem follows immediately from the above definition:

Theorem 5.2.1 *L is accepted by an s-AFA with k states iff L^R is accepted by an rs-AFA with k states.*

An rs-AFA with k states (including the final state) can be represented by k Boolean variables $x_0, \ldots, x_k$ with $k \cdot |\Sigma|$ Boolean functions $x_i(a)(x_1, \ldots, x_k)$ for each $a \in \Sigma$ and i, $0 \leq i \leq k$, and the initial values $x_i = 1$ iff $i \in S$.

Example 5.2.1 *The language $ab^*a + ab^*$ is accepted by the following rs-AFA A:*

$$\begin{array}{ll} x_0(a) = \bar{x}_1\bar{x}_2 + \bar{x}_1 x_2 & x_0(b) = \bar{x}_1 x_2 \\ x_1(a) = x_1 + x_2 & x_1(b) = x_1 + \bar{x}_2 \\ x_2(a) = x_1 + \bar{x}_2 & x_2(b) = 1 \end{array}$$

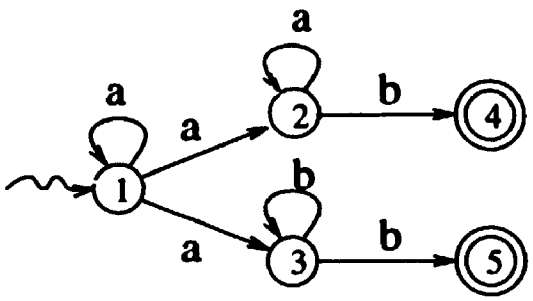
with $I = (0,0,0)$, i.e., the initial values $x_i = 0$, $0 \leq i \leq 2$, and q_0 being the final state. Consider $w = aba$; then

$$g_0(aba, I) = g_0(ab, (1,0,1)) = g_0(a, (1,0,1)) = 1$$

So aba is accepted by the rs-AFA A.

Note that an NFA can be considered as a special type of rs-AFA, such that $g_q(a)$ is a Boolean function with only or (+) operations for each $q \in Q$ and $a \in \Sigma$. We call such an automaton an NFA in the rs-AFA form.

Example 5.2.2 *Let $\Sigma = \{a,b\}$. Consider the following NFA B.*



B can be represented in the rs-AFA form as follows:

$$\begin{aligned} x_0(a) &= x_4 + x_5 & x_0(b) &= x_4 + x_5 \\ x_1(a) &= x_1 & x_1(b) &= 0 \\ x_2(a) &= x_1 + x_2 & x_2(b) &= 0 \\ x_3(a) &= x_1 + x_3 & x_3(b) &= x_3 \\ x_4(a) &= 0 & x_4(b) &= x_2 \\ x_5(a) &= 0 & x_5(b) &= x_3 \end{aligned}$$

with the initial values $x_i = 0$, $0 \leq i \leq 5$.

Theorem 5.2.2 *Let* $A = (Q, \Sigma, S, q_f, g)$ *be an rs-AFA. Then we can construct an equivalent rs-AFA* $A' = (Q, \Sigma, S', g_f, g')$, *such that* $S' = \emptyset$ *(i.e.,* $I = (0, \ldots, 0)$*) if* $\varepsilon \notin L(A)$ *or* $S' = \{g_f\}$ *(i.e.,* $I = (1, 0, \ldots, 0)$*) if* $\varepsilon \in L(A)$

Proof. The function g' is defined in the following two steps:

(1) For each $q \in Q - S$ and $a \in \Sigma$, define $g''_q(a) = g_q(a)$. For each $q \in S$ and $a \in \Sigma$, define $g''_q(a) = \overline{g_q(a)}$.

(2) For each $q \in Q$ and $a \in \Sigma$, we define that $g'_q(a)$ is the same as $g''_q(a)$ except that each x_q, $q \in S$, is replaced by $\overline{x_q}$.

It is easy to prove that $L(A) = L(A')$. □

Note that the above theorem does not hold if we restrict rs-AFA to the ones that are NFA in rs-AFA form.

Given an arbitrary rs-AFA $A = (Q, \Sigma, S, q_f, g)$, we can construct an equivalent NFA $A' = (Q', \Sigma, S', q'_f, g')$ (in rs-AFA form) as follows:

Assume that $Q = \{0, 1, \ldots, k\}$ and $g_f = 0$. Let A be represented by the Boolean variables x_i, $i \in Q$, with the Boolean functions $x_i(a)$, $a \in \Sigma$ and $0 \leq i \leq k$, and the

initialization vector $I^{(A)}$ ($I_i^{(A)} = 1$ iff $i \in S$). Then $Q' = \{0, 1, \ldots, 2^k\}$. Define for each $i \in Q'$ and $i \neq 0$, a variable y_i, by a minterm [24] of the variables $x_1, \ldots, x_k$. i.e.,

$$\begin{aligned} y_1 &\equiv \overline{x_1}\ \overline{x_2}\ \cdots\ \overline{x_{k-1}}\ \overline{x_k} \\ y_2 &\equiv \overline{x_1}\ \overline{x_2}\ \cdots\ \overline{x_{k-1}}\ x_k \\ &\cdots\ \cdots\ \cdots \\ y_{2^k} &\equiv x_1\ x_2\ \cdots\ x_{k-1}\ x_k \end{aligned}$$

and $y_0 = x_0$. Then for each $a \in \Sigma$ and $i \in Q'$, the function $y_i(a)$ is defined by substituting each function $x_j(a)$, $i \leq j \leq k$, for x_j in the above definition.

5.3 SEs to rs-AFA

Now we are ready to describe how an rs-AFA is constructed for a given synchronization expression.

- **Sequencing** ($e \equiv e_1 \rightarrow e_2$)

  Given
  $A = (Q_A, \Sigma, S_A, q_{f_A}, g_A)$: an m-state rs-AFA represented by the m variables $x_0, x_1, \ldots, x_m$ and the initialization vector $I^{(A)}$, such that $L(A) = L(e_1)$,
  $B = (Q_B, \Sigma, S_B, q_{f_B}, g_B)$: an n-state rs-AFA represented by the n variables $y_0, y_1, \ldots, y_n$ and the initialization vector $I^{(B)}$, such that $L(B) = L(e_2)$,
  then we construct C, such that $L(C) = L(e)$ as follows:

  C has $m + n + 1$ states represented by the following set of $m + n + 1$ variables

  $$\{y_0, x_1, \ldots, x_m, \hat{y}_1, \ldots, \hat{y}_n\}$$

  where $x_1, \ldots, x_m$ are the same as in A and for each $a \in \Sigma$ and $0 \leq i \leq n$

  $$\hat{y}_i(a) = \begin{cases} y_i(a) & \text{if } i \notin S_B \\ y_i(a) + x_0(a) & \text{if } i \in S_B \end{cases}$$

- **Join** ($e \equiv e_1 \parallel e_2$)

  Given
  $A = (Q_A, \Sigma_A, S_A, q_{f_A}, g_A)$: an m-state rs-AFA represented by the m variables $x_0, x_1, \ldots, x_m$ and the initialization vector $I^{(A)}$, such that $L(A) = L(e_1)$,
  $B = (Q_B, \Sigma_B, S_B, q_{f_B}, g_B)$: an n-state rs-AFA represented by the n variables

$y_0, y_1, \ldots, y_n$ and the initialization vector $I^{(B)}$, such that $L(B) = L(e_2)$, $\Sigma_A \cap \Sigma_B = \emptyset$,
then we construct C, such that $L(C) = L(e)$ as follows:

C has set of $m + n + 3$ states represented by the following $m + n + 3$ variables

$$\{z_0, x_0, x_1, \ldots, x_m, y_0, y_1, \ldots, y_n\}$$

and the following additional functions:

$$\begin{array}{lll} x_i(a) & = x_i & \text{for } a \in \Sigma_B,\ 0 \leq i \leq n \\ y_i(a) & = y_i & \text{for } a \in \Sigma_A,\ 0 \leq i \leq n \\ z_0(a) & = x_0(a)y_0(a) & \text{for } a \in \Sigma_A \cup \Sigma_B \end{array}$$

- **Selection** $(e \equiv e_1 \mid e_2)$

Given
$A = (Q_A, \Sigma, S_A, q_{f_A}, g_A)$: an m-state rs-AFA represented by the m variables $x_0, x_1, \ldots, x_m$ and the initialization vector $I^{(A)}$, such that $L(A) = L(e_1)$,
$B = (Q_B, \Sigma, S_B, q_{f_B}, g_B)$: an n-state rs-AFA represented by the n variables $y_0, y_1, \ldots, y_n$ and the initialization vector $I^{(B)}$, such that $L(B) = L(e_2)$,
then we construct C, such that $L(C) = L(e)$ as follows:

C has $m + n + 1$ states represented by the following set of $m + n + 1$ variables

$$\{z_0, x_1, \ldots, x_m, y_1, \ldots, y_n\}$$

where $x_1, \ldots, x_m$ are the same as in A, $y_1, \ldots, y_n$ are the same as in B, and for each $a \in \Sigma$ and $0 \leq i \leq n$

$$z_0(a) = x_0(a) + y_0(a)$$

- **Repetition** $(e \equiv e_1^*)$

Given
$A = (Q_A, \Sigma, S_A, q_{f_A}, g_A)$: an m-state NFA in the rs-AFA form represented by the m variables $x_0, x_1, \ldots, x_m$ and the initialization vector $I^{(A)}$, such that $L(A) = L(e_1)$,
then we construct C, such that $L(C) = L(e)$ as follows:

Without loss of generality, we assume $S_A - \{q_{f_A}\} = \emptyset$. C has $n+1$ states represented by the following set of $n+1$ variables

$$\{\hat{x}_0, \hat{x}_1, \ldots, \hat{x}_n\},$$

and define, for each $a \in \Sigma$ and $0 \leq i \leq n$. $\hat{x}_i(a)$ to be the same as $x_i(a)$ except that all x's are changed to $\hat{x}$ if the function $x_i(a)$ does not contain any term $\overline{x_j}$ for some j, $1 \leq j \leq n$. Otherwise, $\hat{x}_i(a)$ is defined as $x_i(a) + x_0(a)$ with all x's being changed to $\hat{x}$'s.

- **Intersection** ($e \equiv e_1 \,\&\, e_2$)

  Given
  $A = (Q_A, \Sigma, S_A, q_{f_A}, g_A)$: an m-state rs-AFA represented by the m variables $x_0, x_1, \ldots, x_m$ and the initialization vector $I^{(A)}$, such that $L(A) = L(e_1)$,
  $B = (Q_B, \Sigma, S_B, q_{f_B}, g_B)$: an n-state rs-AFA represented by the n variables $y_0, y_1, \ldots, y_n$ and the initialization vector $I^{(B)}$, such that $L(B) = L(e_2)$,
  then we construct C, such that $L(C) = L(e)$ as follows:

  C has $m+n+1$ states represented by the following set of $m+n+1$ variables

  $$\{z_0, x_1, \ldots, x_m, y_1, \ldots, y_n\}$$

  where $x_1, \ldots, x_m$ are the same as in A, $y_1, \ldots, y_n$ are the same as in B, and for each $a \in \Sigma$ and $0 \leq i \leq n$

  $$z_0(a) = x_0(a) \cdot y_0(a)$$

5.4 Implementation Environment

Our implementation is based on a Sequent Symmetry multiprocessor. The hardware configuration of the Sequent Symmetry S27 multiprocessor hosted by the Department of Computer Science, The University of Western Ontario, is as follows:

- Ten 16-MHz Intel 80386 32-bit CPUs. Six of these CPUs have Weitek Floating Point Accelerator for better floating point performance.
- 48 Mbytes of shared memory with 500 Mbytes of virtual memory space.

- 64 Kbytes, two-way set-associative, write-back cache on each CPU. They allow the CPU to operate most of the time within its own memory environment, reducing the frequency of bus access to the shared memory.

- A 64-bit system bus, which carries data among the system's CPUs, memory modules, and peripheral subsystems; it supports pipelined I/O and memory operations and variable-size data packets, and has a channel bandwidth of 80 Mbytes per second.

The software support for parallel programming on Sequent Symmetry consists of:

- **DYNIX operating system.** DYNIX is a Sequent version of UNIX compatible with both UNIX 4.2bsd and UNIX System V. In addition to the operating system calls typically found in a UNIX system, DYNIX provides a set of routines to facilitate parallel processing. The C language accompanying the Sequent consists of simple low-level extensions of the sequential language with operating system calls to the parallel library. The extensions are very low-level.

- **Shared memory.** The DYNIX operating system allows two or more processes to share common regions of the system memory. Any process that has access to a shared-memory region can read or write in that region in the same way it reads or writes in an ordinary memory. The keyword **shared** placed before a global variable declaration indicates that all processes are to share a single instance of that variable.

- **Parallel programming library.** A library of routines, designed for use with C, Fortran, or Pascal, allows the use of shared memory and mutual exclusion mechanisms and supports the most commonly used parallel programming mechanisms, including such low-level synchronization mechanisms as spinlocks, barriers, and fork/join operations.

5.5 Overview of the Implementation: A Example

The translation of a ParC program with synchronization expressions into a C program with system calls consists of the following steps:

1. The syntax of the ParC special constructs, that is shared variables, PEXEC, PFOR, and SFOR statements, tags, and synchronization expressions, is checked by the ParC parser, which has been built using Yacc and Lex. Error messages are generated for the syntax errors that occur in the ParC special constructs.

2. The statements *PEXEC*, *PFOR*, and *SFOR* are translated into low-level Sequent system calls and C code.

3. A table of statement tags is built, and all tags are checked for their scope and declaration.

4. For synchronization expressions, corresponding AFA are constructed, as described in the previous section.

5. The AFA are translated into corresponding C programs and Sequent system calls, which are inserted into the proper places of the program, according to the tag table.

6. Finally, the program generated by the preprocessor is sent to the Sequent C compiler for compilation.

In the following, we provide a simple example to show how a ParC program with synchronization expressions is translated into a C program with the Sequent system calls. For a better understanding of the major points of the implementation, we omit some unimportant details, and use a DFA instead of an AFA.

```
main(){
    shared int w;
    int k, r;
    tag a, b;
    int f();
    restrict a->b;

    pexec{
        a:: w = f(k);
        b:: r = w;
    }
}
```

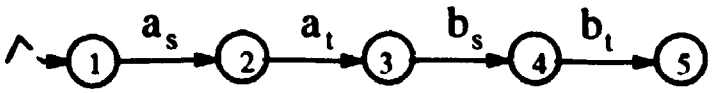


Figure 5.1: a DFA for the SE: a-¿b

In order to explain our idea clearly without looking at the implementation details, we define a macro

```
wait_and_set( int  state; int i; int j )
```

It waits until the condition `state == i` is met, and then it sets `state = j`. The macro is an indivisible (exclusive) operation and can be implemented in many different ways.

The synchronization language for the synchronization expression `restrict a->b` is $\{a_s a_t b_s b_t\}$; this language is accepted by a 5-state DFA, which is shown in Figure 5.1.

The following is the C program translated from the given ParC program. The synchronization constraints specified by the synchronization expressions have been incorporated into the program.

```
#include<parallel/microtask.h>
#include<parallel/parallel.h>
#include<sys/wait.h>
#include<sys/types.h>
#include<stdio.h>

main() {
shared int w;
int k, r;
shared int _state=0;
int f();

    {
    int _pw, _pid;

        if ((_pid=fork()) == 0){
            wait_and_set(_state, 0, 1);
```

```
                r = w;
            wait_and_set(_state, 1, 2);
            exit(0);
        }
        else if (_pid < 0){
            printf("fork error\n");
            exit(1);
        }
        wait_and_set(_state, 2, 3);
        w = f(k);
        wait_and_set(_state, 3, 4);
        _pw = wait(0);
        if (_pw < 0){
            printf("execution error within pexec\n");
            exit(1);
        }
    }
}
```

For more implementation details of *ParC*, please refer to the manual pages for the pp command, which we have developed for the Sequent system of the Department of Computer Science, the University of Western Ontario.

Chapter 6

Conclusion and Future Directions

The main contributions of this thesis may be summarized as follows:

- We have developed high-level language constructs for expressing synchronization constraints. The new constructs, synchronization expressions, relieve programmers of the burden of imposing synchronization, requiring them only to specify the necessary constraints. They also provide simple and comprehensible solutions to many synchronization problems, and can specify a wide range of synchronization requirements, at both fine-grain and coarse-grain levels. Solutions that use synchronization expressions to synchronization problems related to both mutual exclusion and events ordering are straightforward and easy to understand. In addition, synchronization expressions do not change the structure of the base language.

- We have introduced a new family of languages called synchronization languages which we use to give a precise semantic description of synchronization expressions. Under this model, relations such as equivalence and inclusion between synchronization expressions can be easily understood and tested. In practice, synchronization languages also provide us with a systematic approach to the implementation and simplification of synchronization expressions in parallel programming languages. Moreover, the study of synchronization languages is, in itself, an interesting topic in formal language theory.

- We have used reversed alternating finite automata in the implementation of synchronization expressions. By using reversed alternating finite automata, we have reduced the number of states which are needed for accepting synchronization languages. Operations in terms of synchronization expressions have

been constructed for reversed alternating finite automata. This method is best suited for implementing synchronization expressions in a shared memory multiprocessor environment, which requires fewer locks and possibly less memory space.

- We have implemented the parallel language constructs *PEXEC*, *PFOR*, and *SFOR* in our parallel programming language *ParC*. *ParC* is a C-based parallel programming language we implemented on a shared-memory multiprocessor Sequent with 10 processors. Constructs *PEXEC*, *PFOR*, and *SFOR* have been tested and have been used for teaching Parallel Programming and Parallel Algorithms courses since 1993. The language construct ***shared***, which can be used to specify shared variables with scopes as described in Chapter 3, has been implemented and tested with *PEXEC*, *PFOR*, and *SFOR* constructs. We also implemented basic synchronization expressions in *ParC*. These basic synchronization expressions do not include the index range and the guard described in previous chapters. These further implementation efforts still need to be made.

6.1 Future directions

The work we present in this thesis involves the development of new language constructs to express synchronization constraints in parallel programming. The semantic model for synchronization languages may be generalized as a model for parallel and distributed computing. Under this framework, some further research topics are:

1. A formal model for the evaluation of the power of synchronization mechanisms. Currently, the power of synchronization mechanisms is still demonstrated by showing solutions to a number of synchronization problems. The limitation of any synchronization mechanism can only be found by examining certain examples. There is no standard set of problems, however, to be used as a basis for the comparison of these mechanisms. It is clear that a well-defined methodology is needed.

2. In chapter 4, we conjecture that for every $L \in \mathcal{L}_{wc}(\Sigma)$, the $R(\Sigma)$-closure of L, i.e., $\Delta^*_{R(\Sigma)}(L)$, is also in $\mathcal{L}_{wc}(\Sigma)$. We also conjecture that the family of Σ-synchronization languages consists of exactly all languages $\Delta^*_{R(\Sigma)}(L)$ where $L \in \mathcal{L}_{wc}(\Sigma)$; that is, the Σ-synchronization languages can be characterized as the

$R(\Sigma)$-closure of languages defined by well-formed consistent regular expressions. Formal proofs are required, and further properties of synchronization languages should be studied from the aspect of formal languages.

3. The implementation of synchronization expressions presently relies on shared-memory multiprocessors. The idea of synchronization expressions can be easily applied to parallel systems with distributed-memory. However, the use of reversed alternating finite automata in implementing synchronization expressions on distributed memory parallel systems needs to be carefully studied.

REFERENCES

[1] I. J. Aalbersberg and G. Rozenberg, "Theory of Traces", *Theoretical Computer Science* 60, (1988) pp.1-82.

[2] W. Abu-Sufah and A. D. Malony, "Vector Processing on the Alliant FX/8 Multiprocessor", *Proceedings 1986 Int. Conf. on Parallel Processing* (Aug.1986), pp. 559-566.

[3] W. B. Ackerman, "Data Flow Languages", *IEEE Computer* Vol.15, No.2, (Feb. 1982), pp.15-25.

[4] T. Agerwala and M. Flynn, "Comments on Capabilities, Limitations and 'Correctness' of Petri Nets", *Proceedings of the ACM First Annual Symposium on Computer Architecture* (1973), pp.81-86.

[5] A.V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principle, Techniques, and Tools*, Addison-Wesley, (1986).

[6] S. Ahuja, N. Carriero and D. Gelernter, "Linda and Friends", *IEEE Computer* (August 1986), pp.26-34.

[7] G. Allan, etc., "The NYU Ultracomputer—Designing an MIMD Shared Memory Parallel Computer", *IEEE Transactions on Computers* Vol. C-32, No.2, (Feb. 1983) pp.175-189.

[8] J. R. Allan and K. Kennedy, "Automatic Translation of Fortran Programs to Vector Form", *ACM TOPLAS* 9 (1987), pp. 491-542.

[9] F. Allan, etc., "An Overview of the PTRAN Analysis System for Multiprocessing", *Proc. of 1987 Int. Conf. on Supercomputing* (1987).

[10] G. T. Almes, "The Impact of Language and System on Remote Procedure Call Design", *Proceedings of the 6th International Conference on Distributed Computing Systems* (May 1986), pp.414-421.

[11] S. Andler, "Predicate Path Expressions", *Proc. 6th ACM Symposiums on Principles of Programming Languages* (1979), pp.226-236.

[12] G. R. Andrews, "Synchronizing Resources", *ACM Transactions on Programming Languages and Systems* 3(4), pp.405-430, 1981.

[13] G. R. Andrews and F. B. Schneider, "Concepts and Notations for Concurrent Programming", *Computing Surveys* 15(1) (1983), pp.3-43.

[14] J. Backus, " Can programming be Liberated from the Von Neumann Style? A Functional Style and Its Algebra of Programs", *Communications of ACM* 21, 8 (Aug. 1978), pp.613-641.

[15] J. C. M. Baeten and W. P. Weijland, *Process Algebra*, Cambridge University Press, Cambridge 1990.

[16] H. E. Bal and A. S. Tanenbaum, "Distributed Programming with Shared Date", *Proceedings of the IEEE CS 1988 International Conference on Computer Languages* (Oct. 1988), pp.82-91.

[17] J. A. Bergstra and J. W. Klop, "Algebra of Communicating Processes with Abstraction", *Theoretical Computer Science* 37(1) (1985) pp.77-12.

[18] A. D. Birman and T. A. Joseph, "Reliable Communication in the Presence of Failures", *ACM Transactions on Computer Systems* 2, 2 (April 1987), pp.234-238.

[19] A. D. Birrell and B. J. Nelson, "Implementing Remote Procedure Calls", *ACM Transactions on Computer Systems* 2, 1 (Feb. 1984), pp.39-59.

[20] T. Bloom, "Evaluating Synchronization Mechanisms", *Proc. 7th ACM Symposiums on Operating Systems Principles*, New York (1979) pp.24-32.

[21] G. Boudel and I. Castellani, "On the Semantics of Concurrency: Partial Orders and Transition Systems", *Proc. TAPSOFT'87, Lecture Notes in Computer Science* Vol.249 (1987), pp.123-137.

[22] S. D. Brookes, C. A. R. Hoare and A. W. Roscoe, "A Theory of Communicating Sequential Processes", *JACM*, Vol.31, No.3, July, 1984, pp.560-599.

[23] D. Bruschi, G. Pighizzini and N. Sabadini, "On the Existence of Minimum Asynchronous Automaton and on Decision Problems for Unambiguous Regular Trace Languages" *Lecture Notes in Computer Science* 294 (1988) pp 334-345.

[24] J. A. Brzozowski and M. Yoeli, *Digital Networks*, Prentice-Hall, (1976).

[25] R. H. Campbell and A. N. Habermann, "The Specification of Process Synchronization by Path Expression", *Lecture Notes in Computer Science*, Vol.16 (1974) pp.89-102.

[26] R. H. Campbell and R. B. Kolstad, "Path Expression in Pascal", *Proc. 4th Int. Conf. on Software Engineering*, (1979) pp.212-219.

[27] R. H. Campbell and R. B. Kolstad, "Practical Applications of Path Pascal in System Programming", Technical Report 217-333-0215, Univ. of Ill., Urbana-Champaign, (1979).

[28] N. Carriero and D. Gelernter, "Linda in Context", *Communications of the ACM*, Vol.32, No.4 (1989) pp.444-458.

[29] A. K. Chandra, D. C. Koren, and L. J. Stockmeyer, "Alternation", *JACM*, 28 (1981) pp.114-133.

[30] von V. Claus, "Zuweisungs-Programme", *Oberwolfach Conference on Formal Languages and Progammming Languages*, (Aug. 1971).

[31] von V. Claus, "Ein Vollständigkeitssatz für Programme und Schaltkreise", *Acta Informatica*, 1 (1971), pp.64-78.

[32] D. Coleman, R.M. Gallimore, J.W. Hughes and M. S. Powell, "An Assessment of Concurrent Pascal", *Concurrent Programming*, Ed. N. Gehani and A. D. McGettrick, Addison-Wesley, (1988).

[33] S. Crespi-Reghizzi and D. Mandrioli, "Petri Nets and Szilard languages", *Information and Control*, 33, 2 (Feb. 1977), pp.177-192.

[34] P. J. Courtois, F. Heymans and D. L. Parnas, "Concurrent Control with 'Readers' and 'Writers' ", *Communications of the ACM* Vol.14 No.10 (Oct. 1971) pp.667-668.

[35] W. Crowther, etc., "Performance Measurements on a 28-node Butterfly Parallel Processor", *Proceedings of the 1985 International Conference on Parallel Processing*, (Aug. 1985), pp.531-540,

[36] J. B. Dennis and E. C. van Horn, "Programming Semantics for Multiprogrammed Computations", *Communications of the ACM*, 9(3) pp.143-155, (1966).

[37] E. W. Dijkstra, "Cooperating Sequential Processes", in *Programming Languages*, ed. F. Genuys, Academic Press, NY, (1968).

[38] E. W. Dijkstra, "Hierarchical Ordering of Sequential Processes", *Acta Informatica*, 1, (1971) pp.115-138.

[39] R. Duncan, "A Survey of Parallel Computer Architectures", *IEEE Computer*, (Feb. 1990).

[40] A. Fellah, H. Jürgensen and S. Yu, "Constructions on Alternating Finite Automata", *International Journal of Computer Mathematics*, vol.35 no.3 & 4 (1990), pp.117-132.

[41] M. J. Flynn, "Very High Speed Computing Systems", *Proceedings of IEEE*, Vol.54 (1966), pp.1901-1909.

[42] L. Flon and A. N. Habermann, "Towards the Construction of Verifiable Software Systems", *Proc. ACM Conf. Data, SIGPLAN Not. 8* (1976) pp.141-148.

[43] V. K. Garg and M. T. Ragunath, "Concurrent Regular Expressions and Their Relationship to Petri Nets", Theoretical Computer Science, 96 (1992), pp.285-304.

[44] N. H. Gehani and W. D. Roome, "Concurrent C", *Software Practice and Experience*, 16(9) (1986), pp.821-844.

[45] N. H. Gehani and T. A. Cargill, "Corrent Programming in the Ada Language: the Polling Bias", *Concurrent Programming*, Ed. N. Gehani and A. D. McGettrick, Addison-Wesley, (1988).

[46] D. Gelernter and A. Bernastein, "Distributed Communication via Global Buffer", *Proceedings ACM Symposium on Principles of Distributed Computing*, (Aug. 1982), pp.10-18.

[47] R. Govindarajan, L. Guo, S. Yu, and P. Wang, "ParC Project: Practical Constructs for Parallel Programming Languages" *IEEE Proceedings of the 15th Annual International Computer Software & Applications Conference*, (1991), pp. 183-189.

[48] R. Govindarajan and S. Yu, "Attempting Guards in Parallel: A Data Flow Approach to Execute Generalized Guarded Commands", *the Proceedings of PARLE'91 Parallel Architectures and Languages Europe*, (June 1991), pp.372-389.

[49] L. Guo and S. Yu, "Synchronization Expressions in Parallel Programming Languages" Technical Report #308 1993, Dept. of Computer Science, University of Western Ontario, Canada, (1993).

[50] L. Guo, K. Salomaa and S. Yu, "Synchronization Expressions and Languages", In *Proceedings of 6th IEEE Symposium on Parallel and Distributed Processing*, (Oct. 1994), pp.257-264.

[51] L. Guo, K. Salomaa and S. Yu, "On Synchronization Languages" accepted by *Fundamenta Informaticae.*

[52] B. K. Haddon, "Nested Monitor Calls", *Operating Systems Review* 11 4 (Oct. 1977), pp.18-23

[53] A. N. Habermann, "Path Expression", *Technique. Report*, Dept. of Computer Science, CMU, Pittsburgh, PA, (June 1975).

[54] P. Brinch Hansen, "A Comparison of Two Synchronizing Concepts", *Acta Informatica*, 1 (1972), pp.190-199.

[55] P. Brinch Hansen, "Structured multiprogramming", *Communications of the ACM*, Vol.15 No.7 (Jul. 1972), pp.574-578.

[56] P. Brinch Hansen, *"Operating System Principles"*, Prentice-Hall, Englewood Cliffs, N. J., (1973).

[57] P. Brinch Hansen, "The Programing Language Concurrent Pascal", *IEEE Transactions on Software Engineering*, Vol.SE-1 No.2 (Jun. 1975), pp. 199-207.

[58] P. Brinch Hansen, "Distributed Processes: A Concurrent Programming Concept", *Communications of the ACM*, 21(11), (1978), pp.934-941.

[59] P. Brinch Hansen, "Monitor and Concurrent Pascal: A Personal History", *Proceedings of ACM History of Programming Languages Conference*, (Apr. 1993), pp.1-33.

[60] P. J. Hatcher and M. J. Quinn, *Data-parallel Programming on MIMD Computers*, The MIT Press, (1991).

[61] P. B. Henderson and Y. Zalcstein, "Synchronization Problems Solvale by Generalized PV Systems", *Journal of the ACM*, Vol.27 No.1 (Jan. 1980), pp.60-71.

[62] M. R. Headington and A. E. Oldehoeft, "Open Predicate Path Expression and Their Implementation in Highly Parallel Computing Environments", *Proc. of the 1985 Intern. Conf. on Parallel Processing*, (1985), pp.239-254.

[63] M. Hennessy, *Algebraic Theory of Processes*, The MIT Press, Cambridge, Mass. (1989).

[64] W. D. Hillis and G. L. Steele Jr., "Data Parallel Algori'hms", *Communications of the ACM*, 29 12 (Dec. 1986), pp.1170-1183.

[65] C. A. R. Hoare, "Towards a Theory of Parallel Programming", *Operating Systems Techniques*, ed. C. A. R. Hoare and R. H. Perrot, pp.61-71, Academic Press, NY, (1972).

[66] C. A. R. Hoare, "Monitors: An Operating System Structuring Concept", *Communications of the ACM*, 17(10) (1974), pp.549-557.

[67] C. A. R. Hoare, "Communicating Sequential Processes", *Communications of the ACM*, 21(8) (1978), pp.666-677.

[68] J. H. Howard, "Proving monitors", *Communications of the ACM*, 19(5) (1976), pp.273-279.

[69] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation* Addison-Wesley, (1979).

[70] S. Horvath, E. Kinber, A. Salomaa and S. Yu, "Decision Problems Resulting From Grammatical Inference", *Annales Academiae Scientiar Fennicae*, Series A. 1. Mathematica Vol.12 (1987), pp.287-298.

[71] T. Hu, "Parallel Sequencing and Assembly Line Problems", *Operation Research*, Vol.9, (1961), pp.841-848.

[72] IMMOS Ltd., "Occam Programming Manual", Prentice-hall, Englewood Cliffs, NJ, (1984)

[73] H. F. Jordan, "HEP Architecture, Programming and Performance", *Parallel MIMD Computation: the HEP Supercomputer and Its Applications*. J. Kowalik, ed., MIT Press, Cambridge, Mass., (1985), pp.1-40.

[74] W. H. Kaubisch, R. H. perrott, and C. A R. Hoare, "Quasiparallel Programming", *Software Practics Experience*, 6, (1976), pp.341-356.

[75] J. L. Keedy, "On Structuring Operating Systems with Monitors", *ACM Operating Systems Review*, 13(1), (1979), pp.5-9.

[76] J. L. W. Kessels, "An alternative to Event Queues for Synchronization in Monitors", *Communications of the ACM*, 20(7) (1977), pp.500-503.

[77] R. B. Kieburtz and A. Silberschatz, "Comments on 'Communicating Sequential Processes' ", *Concurrent Programming*, Ed. N. Gehani and A. D. McGettrick, Addison-Wesley, (1988).

[78] S. R. Kosaraju, "Limitations of Dijkstra's Semaphore Primitives and Petri Nets", *Operating Systems Review*, 7, 4 (Oct. 1973) pp.122-126.

[79] S. Lafortune and H. Yoo, "Some Results on Petri Net Languages", *IEEE Transactions on Automatic Control*, Vol. 35, Iss 4 (1979), pp.482-485.

[80] M. S. Lam and M. C. Rinard, "Coarse-Grain Parallel Programming in Jade" Proc. of *3rd ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, (Apr. 1991), pp.94-1051.

[81] B. W. Lampson and D. D. Redell, "Experience with Processes and Monitors in Mesa", *Communications of ACM*, 23, 2, (Feb. 1980), pp.105-117.

[82] P. E. Lauer and R. H. Campbell, "Formal Semantics of a Class of High Level Primitives for Coordinating Concurrent Processes", *Acta Information* 5, (1975), pp.297-332.

[83] P. E. Lauer and M. W. Shields, "Abstract Specification of Resource Accessing Disciplines: Adequacy, Starvation, Priority and Interrupts." *SIGPLAN Notice* 13, (1978) pp.41-59.

[84] P. E. Lauer, P. R. Torrigiani, and M. W. Shields, "COSY—A System Specification Language Based on Paths and Processes", *Acta Inform.* 12 (1979) pp.109-158.

[85] P. E. Lauer, M. W. Shields and E. Best, "Design and Analysis of Highly Parallel and Distributed Systems", *Lecture Notes in Computer Science*, 86, (1980), pp.451-503.

[86] A. Lister, "The Problem of Nested Monitor Calls", *Operating Systems Review*, 11, 3, (Jul. 1977), pp.5-7.

[87] A. Lister and P. Sayer, "Hierarchical Monitors", *Software-Practice and Experience*, 7, (1977), pp.613-623.

[88] M. Maekawa, A. Oldehoeft and R. Oldehoeft, *Operating Systems: Advanced Concepts*, Benjamin/Cummings, (1987).

[89] A. Mazurkiewicz, "Trace Theory", *Lecture Notes in Computer Science* 255, Springer (1986).

[90] R. Milner, "A Calculus of Communicating Systems", *Lecture Notes in Computer Science*, Vol. 92, Springer, Berlin, (1980).

[91] R. Milner, "Calculi for Synchrony and Asynchrony", *Theoretical Computer Science*, 25(1983), pp.267-30.

[92] R. Milner, "Lectures on a Calculus for Communicating Systems", Seminar on Concurrency, *Lecture Notes in Computer Science*, 13 (1981), pp.197-220

[93] T. Murate, "Petri Nets - Properties, Analysis and Applications", *Proceedings of the IEEE*, Vol.77, Iss.4 (1989) pp.541-580.

[94] M. Nielsen, G. Plotkin and G. Winskel, "Petri Nets, Event Structures and Domains", *Theoretical Computer Science*, 13 (1981), pp.85-108.

[95] A. Osterhaug, "Guide to Parallel Programming on Sequent Computer Systems", Sequent Computer Systems, Inc., (1990).

[96] D. Padua and M. Wolfe, "Advanced Compiler Optimizations for Supercomputers", *Communications of ACM*, 29 (1986), pp.1184-1201

[97] D. Padua, etc. "Restructuring Fortran Programs for Cedar", *Proc. of 1991 Int. Conf. on Parallel Processing*, (Aug. 1991), pp. I57-I66.

[98] D. L. Parnas, "On a solution to the Cigarette Smokers' Problem without Conditional Statements", *Communications of CACM*, 18(3) (1975), pp.181-183.

[99] D. L. Parnas, "The Non-problem of Nested Monitor Calls", *Operating Systems Review*, 12, 1, (1978), pp.12-14.

[100] G. L. Peterson, "Myths about the mutual exclusion problem", *Information Process Letter*, 1?, 3, (June 1981), pp.115-116.

[101] J. L. Peterson, "Computation Sequence Sets" *J. Computer and System Science*, 13, 1 (Aug. 1976) pp.1-24.

[102] J. L. Peterson, "Petri Nets" *Computing Survey*, Vol.9, No 3, (Sept. 1977). pp.223-252.

[103] J. L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice-Hall, (1981).

[104] J. L. Peterson, *Operating System Concepts*, 2nd Ed., Addison-Wesley, (1987).

[105] A. Petit, "Distribution and Synchronized Automata", *Theoretical Computer Science* 76 (1990) pp.285-308.

[106] M. Raynal and J. Helary, *Synchronization and Control of Distributed Systems and Programs*, John Wiley and Son Ltd., (1990).

[107] G. Roenberg and R. Verraedt, "Subset Languages of Petri Nets", *Theoretical Computer Science*, 26, (1983), pp.301-326; 27, (1983), pp.85-108.

[121] N. Wirth, "Modula: A Language for Modular Multiprogramming", *Software Practice Experience*, 7, (1977), pp.3-35.

[122] D. Wood, "*Theory of Computation*" Harper & Row, (1987).

[123] S. Yu, K. Salomaa, and Q. Zhuang, "On the State Complexity of Some Basic Operations on Regular Languages", accepted by *Theoretical Computer Science* in June 1992.

[124] W. Zielonka, "Notes on Finite Asynchronous Automata" *Theoretical Informtics and Applications* Vol.21, No.2, (1987), pp.99-135.

[108] O. Roubine and J. C. Heliard, "Parallel Processing in Ada", in *On the Construction of Programs*, ed. R.M. McKeag and A. M. Macnaghten, 193-212, Cambridge University Press, MA, (1980).

[109] A. Schiper, *Concurrent Programming*, Halsted Press, (1988)

[110] A. Shaw, "Software Description with Flow Expressions", *IEEE Transactions of Software Engineering SE-4*, 3 (1978) pp.242-254.

[111] P. D. Stotts, "A Comparative Survey of Concurrent Programming Languages", *Concurrent Programming*, Ed. N. Gehani and A. D. McGettrick, Addison-Wesley, (1988).

[112] A. S. Tanenbaum and R. van Renesse, "A Critique of Remote Procedure Call Paradigm", *Proceeding of the EUTECO 88 Conference*, (Apr. 1988), pp.775-783.

[113] A. Trew, and G. Wilson, *"Past, Present, Parallel -A Survey of Available Parallel Computing Systems"*, Springer-Verlag, (1991).

[114] J. Ullman, " NP-Complete Scheduling Problems", *Journal of Computer and System Sciences*, Vol.10 (1975), pp. 384-393.

[115] U. S. Department of Defense, "Programming Language Ada Reference Manual", *Lecture Notes in Computer Science*, Vol.106, (1981).

[116] R. Valk and G. Vidal-Naquet, "Petri Nets and Regular Languages", *J. of Computer and System Sciences*, 23, (1981), pp.299-325.

[117] P. Wegner and S. A. Smolka, "Processes, Tasks and Monitors; A Comparative Study of Concurrent Programming Primitives", *Concurrent Programming*, Ed. N. Gehani and A.D. McGettrick, Addison-Wesley, (1988).

[118] J. Welsh and M. McKeay, *Structured System Programming*, Prentice-Hall, (1980).

[119] C. Wetherell, "Design Considerations for Array Processing Languages", *Concurrent Programming*, Ed. N. Gehani and A.D. McGettrick, Addison-Wesley, (1988).

[120] H. Wettsten, "The Problem of Nested Monitor Calls revisited", *Operating Systems Review*, 12, 1, (Jan. 1978), pp.19-23.